Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE MÁSTER

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DE SISTEMAS Y DE LA COMPUTACIÓN

EVALUACIÓN DE LA TÉCNICA DE PRUEBA DE MUTACIONES PARA NUEVOS ESTÁNDARES DE C++

AUTOR: MIGUEL ÁNGEL ÁLVAREZ GARCÍA

Puerto Real, septiembre 2020

Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE MÁSTER

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DE SISTEMAS Y DE LA COMPUTACIÓN

EVALUACIÓN DE LA TÉCNICA DE PRUEBA DE MUTACIONES PARA NUEVOS ESTÁNDARES DE C++

DIRECTOR: INMACULADA MEDINA BULO
CODIRECTOR: PEDRO DELGADO PÉREZ
AUTOR: MIGUEL ÁNGEL ÁLVAREZ GARCÍA

Puerto Real, septiembre 2020

DECLARACIÓN PERSONAL DE AUTORÍA

Miguel Ángel Álvarez García con DNI 49192381-G, estudiante del Máster en Investigación en Ingeniería de Sistemas y de la Computación en la Escuela Superior de Ingeniería de la Universidad de Cádiz, como autor de este documento académico, titulado EVALUACIÓN DE LA TÉCNICA DE PRUEBA DE MUTACIONES PARA NUEVOS ESTÁNDARES DE C++ y presentado como trabajo final de Máster.

DECLARO QUE

Es un trabajo original, que no copio ni utilizo parte de obra alguna sin mencionar de forma clara su origen, tanto en el cuerpo del texto, como en su bibliografía, y que no empleo datos de terceros sin la debida autorización, de acuerdo con la legislación vigente. Así mismo, declaro que soy plenamente consciente de que no respetar esta obligación podrá implicar la aplicación de sanciones académicas, sin perjuicio de otras actuaciones que pudieran iniciarse.

En Puerto Real, a 9 de septiembre de 2020

Fdo: Miguel Ángel Álvarez García

Agradecimientos

A mi madre, por haber confiado en mí, por cuidarme y ayudarme en todo momento y ser la persona en la que confiar en todos y cada uno de mis días.

A mis abuelos, Antonio y Manoli, por ayudarme en todo lo que han podido, por confiar en mi en todo momento.

A mi tío, por cuidarme, por hacerme ser la persona que he llegado a ser.

A Inma, por su apoyo en todo momento, por su apuesta de futuro en mi y por su ayuda inconfundible cada vez que la he necesitado.

A Pedro, por su ayuda para la puesta en marcha de este TFM, sus múltiples correos y correcciones, sin él, nada de esto podría haber salido adelante.

A la Delegación de Estudiantes de la ESI, mi Dele, mi segunda familia, por estar siempre ahí y haber contado conmigo para volver a ser el Subdelegado de Centro. A todas esas personas que me llevo, muchísimas gracias.

A Juanjo, por su ayuda y su confianza en mí, gracias por todo.

Al resto del centro, su dirección, su personal de administración y servicios, sus coordinadores de grado y máster y sus directores de departamento con los que he tenido relación en algún momento, agradeceros todo lo que me habéis aportado en este camino.

Al resto del cuerpo universitario, su rectorado, su personal de administración y servicios, sus directores generales y de secretariado, a todos sus estamentos, a los que me hayan aportado una u otra cosa, sabiduría, saber relacionarme, hablar en público, organizar actos y eventos, gracias por haberme hecho ser lo que soy.

En definitiva, gracias a todos.

Licencia

Resumen

C++ es uno de los lenguajes de programación más utilizados en la industria y, a partir de C++11, se inauguró una nueva era en la historia de C++, donde se prevé crear una versión revisada del lenguaje cada tres años. Dada la rápida actualización y mejora del lenguaje, así como la acogida gradual en la industria de estos nuevos estándares, este proyecto busca la creación de nuevos operadores de mutación para los estándares C++11 y C++14, mediante la herramienta de prueba de mutaciones MuCPP, con el fin de crear una herramienta bien adaptada a los últimos estándares del lenguaje. Se incluirá un análisis exhaustivo, mediante su prueba en programas reales, de los operadores incluidos en los últimos años en la herramienta y los implementados en el presente proyecto. Por otro lado, dada la gran cantidad de versiones del lenguaje y la necesidad de operadores específicos para determinados desarrolladores, se hace necesaria la inclusión de un método de obtención de operadores externos a la herramienta, por lo que finalmente se realiza la implantación de un sistema de complementos.

Palabras clave

Prueba de mutaciones

Operadores de mutación

MuCPP

Mutation tool

Prueba del software

Prueba de caja blanca

Desarrollo de software

Complementos software

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El presente trabajo fin de máster ha sido desarrollado en colaboración con el Grupo UCASE [1] de Ingeniería del Software en su línea de investigación titulada "Prueba de software en la Industria 4.0", mediante la que se quiere conseguir la integración de su herramienta MuCPP en entornos industriales de la provincia.

1.1. Motivación

La Industria 4.0 está cada vez más presente dentro del tejido empresarial, transformando todos los aspectos relacionados con la producción. En este escenario, en el que las empresas se mueven dentro de un entorno completamente conectado, los sistemas software se vuelven cada vez más complejos.

La prueba de software es un pilar fundamental dentro del desarrollo de cualquier proyecto software a fin de medir y mejorar la calidad de los programas. Esta etapa de pruebas puede suponer sobre un 50 % del coste total de un proyecto [2], tanto económico como en tiempo. Con las nuevas exigencias propias de la Industria 4.0, realizar las diferentes actividades implicadas en esta fase de forma manual y exhaustiva podría suponer un coste incluso mayor. Además, la presencia de errores en el software no detectados antes de su puesta en ejecución puede tener graves consecuencias, especialmente en sistemas críticos.

Dado este contexto, se hace necesaria una evolución hacia la automatización de la prueba de software, incorporando técnicas que permitan incrementar la confianza en que nuestro sistema carece de fallos. Una de las técnicas más potentes en este sentido es la *prueba de mutaciones* [3], que consiste en introducir pequeñas modificaciones sintácticas en el código fuente de un programa dado y observar si dicho cambio es identificado, o no, con las pruebas realizadas. Estos fallos se inyectan en base a unas reglas de transformación predefinidas, que simulan fallos de programación habituales, y que se conocen como *operadores de mutación.*

Con la mejora de las herramientas para que se ajusten a las nuevas necesidades de la industria se conseguirá la necesaria acogida e integración de técnicas avanzadas de prueba, como la prueba de mutaciones, en procesos de prueba reales.

Dado que la investigación en torno a esta técnica es continua desde hace años, aportando entre otros aspectos nuevos operadores de mutación, herramientas para diversos lenguajes y formas de reducir el coste, se hace necesario un seguimiento para evaluar la técnica en nuevos estándares del lenguaje de programación C++, como son C++11 y C++14.

1.2. Objetivos

En la actualidad, el lenguaje de programación más extendido en la industria es C++, un lenguaje de programación diseñado en 1979, y utilizado por primera vez en 1983 fuera de un laboratorio científico. Para este lenguaje existen muy pocas herramientas para la prueba de mutaciones, entre las que se encuentra una herramienta denominada MuCPP, desarrollada en el seno de la Universidad de Cádiz, concretamente por miembros del Grupo de investigación UCASE de Ingeniería del Software (TIC025) [1].

MuCPP incluye diversos operadores de mutación, en concreto, un conjunto de operadores de mutación de los denominados "tradicionales", un conjunto a nivel de clase (respecto de la parte orientada a objetos de dicho lenguaje) y otros operadores no incluidos en las categorías anteriores.

A pesar de que de la herramienta se ha mostrado efectiva en su aplicación en múltiples proyectos [4], se observa una necesidad de actualización constante incluyendo nuevos operadores de mutación aparecidos en la literatura a medida que se incluyen nuevas funcionalidades en las diferentes versiones del lenguaje C++.

Además, a medida que pasa el tiempo, y con el desarrollo de nuevas versiones del lenguaje C++, se hace necesaria una forma de poder construir nuevos operadores de mutación de forma sencilla y dar la posibilidad de creación a los usuarios finales de la herramienta MuCPP. Es por ello que se incluye en la herramienta una nueva forma de creación de operadores mediante complementos. Un complemento o plug-in es una aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de una API.

Por todo ello, el presente proyecto lleva a cabo una actualización e implementación de nuevos operadores de mutación, en concreto se realizarán las siguientes mejoras y actualizaciones:

- Implementación de nuevos operadores aparecidos en la literatura para las versiones C++11 y C++14.
- Análisis de los nuevos operadores de mutación incluidos en los últimos años, comparando algunos de ellos con las conclusiones obtenidas por diferentes artículos en la literatura.
- Inclusión de complementos en la herramienta MuCPP mediante el gestor de complementos Pluga.

1.3. Descripción general del proyecto

El presente proyecto se centra en la prueba de mutaciones, una técnica que consiste en la introducción de pequeñas modificaciones sintácticas en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador. Estos fallos son inyectados en base a reglas de transformación predefinidas que simulan fallos de programación habituales, denominadas *operadores de mutación.*

La prueba de mutaciones es una técnica de caja blanca, siendo esta una prueba del software que consiste en escoger distintos valores de entrada para un programa del que se conoce su código fuente examinando todos y cada uno de sus posibles flujos de ejecución. Estas pruebas son aplicables a diferentes niveles como pueden ser clases o métodos.

En el presente proyecto se implementarán diferentes operadores de mutación para la herramienta MuCPP, la cual, está basada en el lenguaje C++. Este lenguaje tiene múltiples versiones y estándares, siendo el estándar más conocido el ISO/IEC C++, el cual inicialmente especifica los requisitos para las implementaciones con C++98, C++03 y C++11, aportando posteriormente diferentes versiones cada tres años, 2014, 2017 y 2020.

En este punto también es importante destacar que MuCPP es una herramienta creada con Clang, uno de los múltiples subproyectos de LLVM, el cual, es un compilador *frontend* de código abierto para los lenguajes de programación C, C++ y Objetive-C.

LLVM es un proyecto que tiene como finalidad la creación de nuevos compiladores optimizados para cualquier lenguaje de programación, suministrando la infraestructura necesaria para el desarrollo de compiladores actuando como *backend* al tomar el código intermedio que los distintos *frontends* generan para los diversos lenguajes. Gracias a estas tecnologías es posible la implementación de nuevos operadores para generar los mutantes de una manera muy robusta para nuevos estándares de C++.

1.4. Alcance

La herramienta actualizada al final del presente Proyecto Fin de Máster pretende ser empleada en la Industria. Además, tendrá un uso científico, tanto en docencia como en investigaciones que pueda surgir sobre la prueba de mutaciones y que será de gran aportación para la literatura existente sobre este tipo de pruebas.

Para la utilización de la presente actualización será necesaria un ordenador con el sistema operativo Ubuntu 18.04 LTS además de software de fácil instalación.

A la finalización de este proyecto se habrá generado lo siguiente:

- Un estudio sobre los operadores aparecidos en la literatura para C++11 y C++14.
- Conjunto de ficheros de código fuente con la implementación de los nuevos operadores de mutación contemplados en el estudio.
- Diferentes ficheros con código fuente para probar los diferentes operadores operadores indicados anteriormente.
- Un análisis completo de los operadores implementados en los últimos años y en el propio estudio mediante su uso en programas reales.
- Una nueva funcionalidad en MuCPP para la ejecución de operadores mediante complementos.

1.5. Organización del documento

El presente documento se estructura en una serie de capítulos, a través de los se describirán detalladamente todas y cada una de las etapas por las que ha ido pasando el presente proyecto desde sus orígenes hasta su finalización. A continuación se realiza un breve resumen de lo que se tratará en cada capítulo.

- **Capítulo 1.** En este capítulo se realizará una introducción al proyecto, pasando por la motivación para su realización, los objetivos, una descripción general del proyecto y el alcance que se tendrá, además de esta sección en la que se explica la estructura del documento.

- **Capítulo 2.** En este capítulo se contará con la planificación que se ha tenido en la realización del presente proyecto, cuantificando tanto en tiempo como de forma monetaria el coste general del proyecto.
- **Capítulo 3.** En el capítulo tres se realizará una introducción a los conceptos generales. Entre estos conceptos se encuentran el lenguaje C++, incluyendo sus nuevos estándares, las pruebas del software, centrándose en la de mutaciones, el proyecto LLVM, Clang y el árbol de sintaxis abstracta o AST. Por último se verá el estado del arte, describiendo las principales herramientas de prueba de mutaciones que se pueden encontrar en la actualidad.
- **Capítulo 4.** En el capítulo cuatro se realizará una breve introducción a los operadores de mutación que se pueden encontrar en la herramienta MuCPP, incluyendo los operadores tradicionales, de herencia, de polimorfismo y enlace dinámico, de sobrecarga, de manejo de excepciones, de reemplazo, de miscelánea y los incluidos en los últimos años.
- **Capítulo 5.** En este capítulo se explicarán los operadores de mutación para C++11 y C++14 aparecidos en la literatura y que han sido implementados en el presente proyecto, como son los operadores FOR, LMB, FWD e INI.
- **Capítulo 6.** En este capítulo se describen las pruebas realizadas a todos y cada uno de los nuevos operadores implementados para la herramienta MuCPP.
- **Capítulo 7.** En el capítulo 7 se describen las pruebas realizadas a todos y cada uno de los nuevos operadores implementados para la herramienta MuCPP sobre programas reales.
- **Capítulo 8.** En este capítulo se describen las pruebas realizadas a todos y cada uno de los nuevos operadores implementados en los últimos años para la herramienta MuCPP sobre programas reales.
- **Capítulo 9.** En este capítulo se implementará una mejora en la herramienta MuCPP para la inclusión de la posibilidad de ejecución de operadores mediante complementos. Además se creará una guía paso a paso de como crear un operador mediante complementos.
- **Capítulo 10.** En este capítulo se explicarán las conclusiones obtenidas mediante la realización de este proyecto y el trabajo futuro a realizar.
- **Capítulo 11.** En este capítulo se desarrollará un glosario de los acrónimos incluidos en el documento y se indicarán diferentes definiciones.
- **Anexo A.** En este anexo se realiza una explicación sobre la instalación de la herramienta, además de un manual de uso de esta.
- **Anexo B.** En el anexo B se realiza una breve explicación de como instalar los diferentes operadores individuales que se han aportado como adjunto al presente documento, además de como se utilizarán para obtener los diferentes mutantes que genera cada operador.

Capítulo 2

Planificación

2.1. Planificación del tiempo

El presente proyecto se ha realizado conforme a ocho meses de trabajo. Destacar que todas las fases aquí expuestas son desarrolladas durante el transcurso del presente documento.

2.1.1. Fase 1: Investigación sobre la prueba de mutaciones en C++11 y C++14

En este periodo de tiempo se ha investigado sobre la prueba de mutaciones en dichas versiones de C++, incluyendo los últimos operadores expuestos en la literatura.

2.1.2. Fase 2: Desarrollo de operadores de mutación

Tras la etapa anterior, se crean los operadores de mutación que no se encontraban en la herramienta MuCPP pero si han aparecido en la literatura, como son los operadores FOR, LMB, FWD e INI.

2.1.3. Fase 3: Pruebas sobre nuevos operadores

Tras la creación de nuevos operadores comenzamos realizando pruebas de los operadores sobre programas creados específicamente para cada uno de ellos, incluyendo todas las particularidades que pueda tener dicho operador. Cabe destacar que mientras se encontraban en desarrollo han sido probados con multitud de entradas y cambios, obteniendo así los operadores que se buscaban.

2.1.4. Fase 4: Pruebas sobre los operadores creados en los últimos años en programas reales

Etapa en la que se prueban todos y cada uno de los operadores de mutación implementados en los últimos años, haciendo uso de programas reales, con el fin de realizar diferentes análisis y comparativas y obtener conclusiones de dichos operadores.

2.1.5. Fase 5: Inclusión de operadores mediante complementos en MuCPP

Etapa en la que se implementa los complementos para la creación de operadores de mutación en la herramienta MuCPP, contemplando la investigación sobre como realizarlo, la instalación de la herramienta Pluga y la implementación en los ficheros fuente de MuCPP.

2.1.6. Fase 6: Documentación

En esta etapa se termina de desarrollar el presente documento, ya que ha sido desarrollado durante todas las etapas anteriores de forma paralela.

2.1.7. Diagrama de Gantt

En la presente sección se muestra el diagrama de Gantt con el transcurso completo del proyecto.

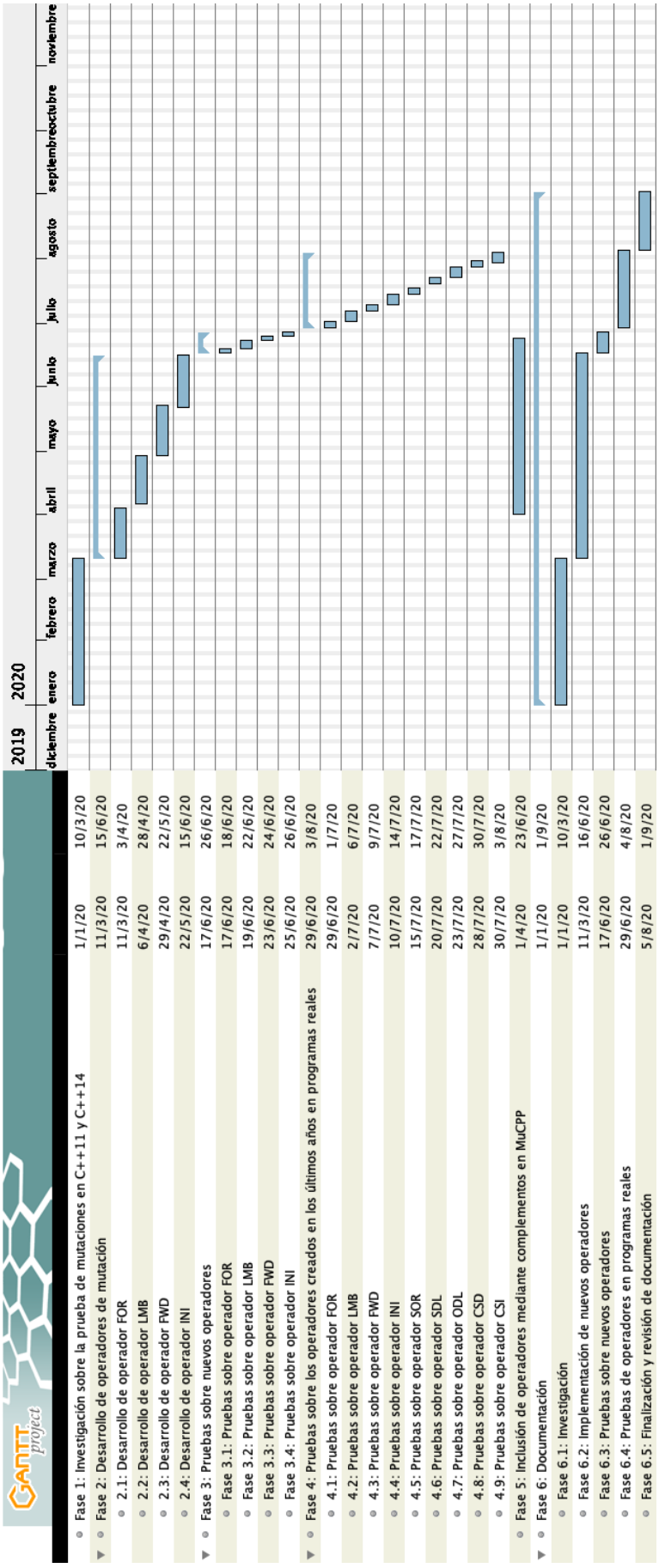

Figura 1: Diagrama de Gantt.

2.2. Coste general del proyecto

En esta sección se desarrollará un presupuesto del coste total aproximado del proyecto realizado, indicando los días trabajados, de investigación, de desarrollo, de pruebas y de documentación. En la tabla 2.2 pueden observarse los días empleados para cada fin, además de los costes totales de cada parte del proyecto. Destacar que el calculo del coste/día se toma teniendo en cuenta el de un único trabajador como investigador.

Detalle	Días	Coste/Día	Coste total
Investigación sobre la prueba de mutaciones en C++11 y C++14.	50	95 €	4.750 €
Desarrollo de operadores de mutación.	69	95 €	6.555 €
Pruebas sobre nuevos operadores.	8	95 €	760 €
Pruebas sobre los operadores creados en los últimos años en programas reales.	26	95 €	2.470 €
Inclusión de operadores mediante complementos en MuCPP	60	95€	5.700€
Documentación.	20	95 €	1.900 €
Coste total del proyecto.	233	95 €	22.135 €

Tabla 1: Tabla coste general del proyecto.

Para el cálculo total de días se ha realizado una estimación en cuanto a las días de trabajo en cada ámbito del proyecto. A continuación se detallan cada uno de los aspectos contenidos en la tabla.

- **Investigación sobre la prueba de mutaciones en C++11 y C++14**. Se estiman un total de cincuenta días en este aspecto, considerando que se ha tenido que investigar sobre la prueba de mutaciones en C++11 y C++14.
- **Desarrollo de operadores de mutación**. Se estiman un total de sesenta y nueve días en este aspecto, considerando que se ha tenido que desarrollar todos y cada uno de los operadores encontrados en el aspecto anterior.
- **Pruebas sobre nuevos operadores**. Se estiman un total de ocho días en este aspecto ya que se ha tenido que realizar diferentes pruebas para los nuevos operadores desarrollados las cuales se adjuntan en este documento.
- **Pruebas sobre los operadores creados en los últimos años en programas reales**. Se estiman un total de veintiséis días en este aspecto ya que se ha tenido que realizar una gran cantidad de pruebas sobre todos y cada uno de los operadores implementados en los últimos años.
- **Inclusión de complementos en la herramienta**. Se estiman un total de sesenta días en este aspecto ya que se ha tenido que realizar un gran trabajo de investigación, implementar la API de los complementos y realizar una actualización interna de la herramienta.
- **Documentación**. Se estiman un total de cien horas en este aspecto, considerando que se han tenido que escribir una cantidad considerable de páginas además de realizar diferentes revisiones de estas.

Capítulo 3

Conceptos y estado del arte

3.1. El lenguaje C++

C++ [5] es un lenguaje de programación de propósito general diseñado con la intención de extender al lenguaje C con la inclusión del paradigma de la Orientación a Objetos y fue diseñado en 1979 por Bjarne Stroustrup.

Por tanto, este lenguaje es orientado a objetos, aunque no "puro", dado que soporta otros estilos de programación como el estructurado. Por ello, también se le denomina lenguaje híbrido. En palabras de su creador, el hecho de que C++ no sea un lenguaje de orientación a objetos puro es más una ventaja que un inconveniente, ya que lo hace más versátil y adecuado para un mayor número de aplicaciones. Más tarde se añadieron facilidades de programación genérica, que, junto a los paradigmas de programación estructurada y programación orientada a objetos, se suele decir que c++ es un lenguaje de programación multiparadigma.

Al igual que ocurre con otros lenguajes como C, el lenguaje C++ fue normalizado por el comité X3J16 de ANSI, el cual, se constituyó en 1989 y terminó su trabajo en 1997. La versión normalizada se denominó C++ ANSI. Posteriormente se creó el estándar ISO/IEC C++, mediante el comité *ISO/IEC 14882:2011 Information technology - Programming languages - C++*, publicándolo en Génova en Febrero de 2012. Este estándar especifica los requisitos para la implementaciones del lenguaje de programación C++ con sus primeras versiones C++98, C++03 y C++11. Actualmente este estándar está aportando versiones cada tres años, en concreto, C++14, C++17 y C++20 a las cuales se han adherido la mayoría de fabricantes de compiladores más modernos.

Además de las facilidades proporcionadas por C, C++ proporciona otras muchos conceptos, como tipos de datos adicionales, clases, plantillas, excepciones, espacios de nombres, sobrecarga de operadores, sobrecarga de nombres de funciones, referencias e instalaciones de bibliotecas adicionales. Podemos determinar que el lenguaje C++ ha evolucionado hasta llegar a ser un lenguaje de propósito general ligeramente recortado. La denominación C++ fue propuesta por Rick Mascitti en el año 1983, cuando el lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Antes de este momento se le había denominado como "C con Clases" [6].

C++ es, a día de hoy, uno de los lenguajes más empleados, tal y como recoge su cuarta posición en el ranking de lenguajes TIOBE [7], por detrás de Java, C y Python. Este lenguaje es usado especialmente en la industria de las comunicaciones, aeronáutica, militar, y un largo etcétera.

En este proyecto se hará uso de las versiones C++11 y C++14, que serán descritas en las próximas secciones.

3.2. Los nuevos estándares de C++, C++11 y C++14

C++11 es una versión del lenguaje de programación C++ reemplazó el 12 de agosto de 2011 al anterior C++03. A partir del 18 de agosto de 2014 fue sustituido por la versión C++14 y más tarde por C++17.

Entre las características más importantes insertadas en C++11 y C++14 encontramos la palabra reservada *auto*, los punteros inteligentes, la semántica de movimiento, las funciones anónimas o funciones lambda o la concurrencia. A continuación se introducirá a diferentes mejoras insertadas en estas versiones y su utilidad para, posteriormente, crear operadores de mutación en base a dichas mejoras.

3.2.1. Bucles for basados en rango

C++11 agregó los bucles for basados en rango, los cuales, mediante una declaración for fácil y sencilla permite iterar sobre una lista de elementos. De esta forma, se iterará sobre cada elemento en la lista, trabajando con cualquier tipo que tenga definida una función begin() y end() que retorne iteradores.

Un ejemplo de uso puede ser el que se presenta a continuación:

```
for(Value& v: _values) {
    v--;
}
```

Cabe destacar que en el ejemplo anterior el bucle basado en rango se realiza mediante paso por referencia constante, siendo posible su paso por valor quitando el símbolo &.

Tal y como se expresa en [8], la declaración for basada en rango de C++11 permite procesar todos los elementos de un conjunto sin usar un contador, evitando así la posibilidad de "salirse"de la matriz o vector y eliminando la necesidad de implementar su propio control de límites.

3.2.2. Funciones lambda

C++11 proporciona la capacidad de crear funciones anónimas, llamadas funciones lambda. Estas se definen de una forma sencilla, tal y como podemos ver a continuación.

```
int sum = 1;
[ &sum ]() { sum++; cout << sum << endl; };
```

Estas funciones se definen localmente dentro de otras funciones y pueden capturar (por valor o por referencia) las variables locales de la función que las encierra y luego manipular estas variables en el cuerpo de lambda. Esto se realiza mediante el empleo de los corchetes.

Por otro lado, estas funciones pueden ser asignadas a un nombre y ser llamadas posteriormente con su lista de parámetros tal y como podemos observar a continuación.

```
auto funcionLambda = []( int x ) { cout << x * 2 << endl; };
funcionLambda( 10 );
```

La llamada anterior imprimiría por pantalla el número 20.

3.2.3. Inicialización con listas

C++11 añade la funcionalidad de inicialización mediante listas gracias a su funcion std::initializer_list. Gracias a esta nueva funcionalidad los elementos de un vector también pueden inicializarse en la declaración del vector siguiendo el nombre de la misma con un signo igual y una lista de inicialización separados por comas y delimitados por llaves. A continuación se presenta un programa que utiliza una lista de inicialización para inicializar un conjunto entero con tres valores.

```
array< int, 3 > n = { 1, 2, 3};
```

Si hay menos inicializaciones que elementos del conjunto, los elementos restantes del conjunto se inicializan a cero.

3.2.4. Inferencia de tipos

C++11 permite inicializar una variable sin conocer el tipo al que pertenece, gracias a la inicialización explícita con la palabra clave *auto*. Esto crea una variable del tipo específico del inicializador. A continuación se puede observar un ejemplo muy básico y que cualquier programador puede entender.

```
auto var = 1;
```

El tipo de var está bien definido, pero es más fácil de determinar por el usuario. Es un int, el cual es el mismo tipo que el literal de un número entero.

auto es también útil para reducir la verbosidad del código, consiguiendo no usar un gran número de palabras a la hora de determinar el tipo concreto.

3.2.5. Los constructores de movimiento

Antes de C++11, las variables temporales, llamadas *rvalues* a menudo se encuentran en el lado derecho de una asignación. Estos fueron pensados para nunca ser modificados aunque en algunos casos los temporales habrían podido ser modificados.

C++11 agrega un nuevo tipo de referencia no constante llamada referencia rvalue, identificada por T&&. Esto refiere a los temporales que permite que sean modificados después de que son inicializados, con el propósito de permitir la semántica de movimiento. Un constructor de movimiento de std::vector<T> que toma una referencia rvalue a un std::vector<T> puede simplemente copiar el puntero a la variable interna, de estilo C, fuera del rvalue en el nuevo std::vector<T> y cambiar el puntero dentro del rvalue a null.

Las referencias de rvalue pueden proporcionar beneficios de rendimiento al código existente sin la necesidad de realizar cualquier cambio fuera de la biblioteca estándar. El tipo del valor retornado de una función que retorna un std::vector<T> temporal no necesita ser cambiado explícitamente a std::vector<T>&& para invocar al constructor de movimiento, pues los temporales son considerados automáticamente como rvalues. También, las referencias rvalue solamente pueden ser modificadas bajo ciertas circunstancias, siendo pensado para ser usado inicialmente con los constructores de movimiento.

Debido a la naturaleza de las referencias rvalue, y a una cierta modificación para las referencias lvalue (referencias regulares), las referencias rvalue permiten a los desarrolladores proporcionar forwarding de funciones perfectos. Cuando se combina con las plantillas variadic, esta capacidad permite plantillas de función que pueden perfectamente remitir (forward) argumentos a otra función que tome esos argumentos particulares. Esto es más

útil para forwarding de parámetros del constructor, para crear funciones que llamarán automáticamente al constructor correcto para los argumentos correctos.

3.2.6. Deducción del tipo retorno de una función

C++14 permite deducir el tipo de retorno en todas las funciones. Con el fin de deducir el tipo de retorno, la función debe ser declarada con auto como el tipo cambiado. Un ejemplo básico de esta funcionalidad es la que encontramos en el código siguiente

```
auto retorno();
```

Tal y como se puede observar el tipo de retorno se deduce de la propia función a la que se llama. Si múltiples expresiones de retorno se utilizan en la implementación de la función, todas deben deducir el mismo tipo.

3.2.7. Lambdas genéricas

En las funciones lambda de C++11 los parámetros tienen que ser declarados con tipos concretos. C++14 relaja este requerimiento, permitiendo que los parámetros de funciones lambda puedan ser declarados con la especificación de tipo auto, consiguiendo así no determinar el tipo concreto de los parámetros.

Un ejemplo sencillo puede ser el que encontramos en las siguientes líneas de código.

```
float numero1 = 3.0f;
double numero2 = 3.0;
auto lambda = [](auto x, auto y) {return x + y;};
double resultado = lambda(numero1, numero2);
```

Tal y como se puede observar, el código anterior proporciona una función para cualquier par de variables, ya sean del mismo tipo o de diferente tipo, devolviendo la suma de ambos parámetros.

3.3. La prueba de software

Una de las fases más importantes y complejas en el desarrollo de un software es la denominada como prueba del software [9,10], la cual es un "filtro" para todo el desarrollo. Este proceso sirve para descubrir errores y defectos a fin de poder eliminarlos. Existen diferentes definiciones para el concepto de pruebas del software, entre los que se encuentra el definido por Glenford J. Myers [11]:

> ***Las pruebas son el proceso de ejecución de un programa con la intención de encontrar errores.***

No es posible asegurar que un software es totalmente correcto, ya que, no es posible realizar una prueba exhaustiva analizando todas y cada una de las posibles situaciones a las que el software se va a enfrentar en el futuro. Los fallos pueden ser de diferentes tipos como errores de precisión, en las prestaciones, en la documentación o incluso en los procedimientos seguidos durante el desarrollo, que dificultarán el mantenimiento del sistema. Una estrategia de prueba de software proporciona una guía que describe los pasos a realizar como parte de dicha prueba, además del esfuerzo, tiempo y recursos que requerirá [9,10].

Existen múltiples clasificaciones de pruebas del software, entre las que encontramos las pruebas por su ejecución, según lo que verifican o por su enfoque, las cuales tienen diferentes pruebas, como son:

- Por su ejecución:
  - **Pruebas manuales.** Las pruebas manuales son aquellas ejecutadas por una o mas personas simulando las acciones de un usuario final.
  - **Pruebas automáticas.** Las pruebas automáticas son aquellas ejecutadas mediante scripts, sin supervisión simultanea, por lo que son realizadas más rápidamente.
- Por lo que verifican:
  - **Pruebas funcionales.** Las pruebas funcionales son aquellas creadas para comprobar las funcionalidades previamente diseñadas.
  - **Pruebas no funcionales.** Las pruebas no funcionales son aquellas que no comprueba la función que realiza el software, por ejemplo, usabilidad, portabilidad o eficiencia.
- Por su enfoque:
  - **Pruebas de caja blanca.** Las pruebas de caja blanca son aquellas que se basan en el acceso al código fuente, contemplando todos los escenarios posibles.
  - **Pruebas de caja negra.** Las pruebas de caja negra son aquellas que no tienen acceso al código fuente. Se basan en la introducción de diferentes entradas, comprobando que sus salidas son las esperadas.

A continuación se describirán las clasificaciones más utilizadas, como son las pruebas funcionales y las no funcionales o las pruebas de caja blanca y pruebas de caja negra.

3.3.1. Las pruebas funcionales y no funcionales del software

Entre las pruebas funcionales del software encontramos aquellas pruebas que se basan en la ejecución, revisión y retroalimentación de las funcionalidades del software en cuestión. Entre estas pruebas encontramos distintos tipos [10]:

- **Pruebas unitarias.** Las pruebas unitarias son aquellas pruebas que dirigen los esfuerzos de verificación a la unidad más pequeña del diseño del software, centrándose en detectar errores de datos, lógica o algoritmos.
- **Pruebas de integración.** Las pruebas de integración son aquellas pruebas que se realizan para comprobar que los elementos unitarios se comportan de manera adecuada en conjunto. Existen dos tipos de pruebas de integración:
  - **Incremental.** Las pruebas de integración incrementales se llevan a cabo probando cada módulo con el conjunto de módulos que ya hayan sido probados. En este caso, existen diferentes estrategias, como son: la integración descendente, la integración ascendente, la prueba de regresión o la prueba de humo.
  - **No incremental.** Las pruebas de integración no incrementales se llevan a cabo probando cada módulo de forma independiente y, posteriormente, se combinan todos para formar el programa completo.

- **Pruebas alpha.** Las pruebas alpha son aquellas realizadas mientras el software se encuentra en desarrollo y que tienen como fin comprobar que el desarrollo se está realizando de manera correcta y satisfaciendo las necesidades del cliente final.
- **Pruebas beta.** Las pruebas beta son aquellas realizadas tras las pruebas alpha y la finalización del desarrollo, haciéndolo en un entorno real y cuyo fin es el de detectar fallos no vistos anteriormente.
- **Pruebas de aceptación.** Las pruebas de aceptación son aquellas a realizar antes de la liberación de nuevas versiones, comprobando que el software cumple con las expectativas del usuario final.
- **Pruebas de regresión.** Las pruebas de regresión son aquellas realizadas con el fin de asegurar que los casos de prueba que ya habían sido probados y fueron exitosos permanezcan así tras una actualización del software.
- **Pruebas de sistema.** Las pruebas de sistema son aquellas que se realizan a nivel global tras las pruebas unitarias y las pruebas de integración.
- **Pruebas de humo.** Las pruebas de humo son aquellas que tienen como fin el evaluar la calidad del software desarrollado, en el que se realizan pruebas del funcionamiento básico antes de ser entregado al usuario final.

Entre las pruebas no funcionales del software encontramos aquellas pruebas cuyo objetivo es la verificación de un requisito que especifica criterios no funcionales como la disponibilidad, accesibilidad, usabilidad, mantenibilidad, seguridad, rendimiento, etc. Podemos clasificar las pruebas no funcionales según el tipo de requisito no funcional que prueba:

- **Pruebas de compatibilidad.** Son aquellas pruebas que verifican el funcionamiento de un determinado software en diferentes entornos.
- **Pruebas de seguridad.** Pruebas que miden la seguridad de un determinado software, estas pruebas pueden ser realizadas antes de su puesta en funcionamiento por el usuario final o posteriormente.
- **Pruebas de stress.** Pruebas que miden el nivel de esfuerzo que se le puede aplicar a un determinado software, determinando así la carga de trabajo que soportará.
- **Pruebas de usabilidad.** Es una prueba centrada en el usuario mediante la cual se pueden observar las interacciones que estos realizan en el software con el fin de mejorarlas.
- **Pruebas de rendimiento.** Son aquellas pruebas en las que se mide el tiempo que tarda el software en realizar determinadas tareas en un entorno.
- **Pruebas de escalabilidad.** Pruebas que se realizan para comprobar el nivel de escalabilidad que posee un determinado software, o en otras palabras, cuanta demanda puede soportar sin realizar grandes cambios en su configuración.
- **Pruebas de mantenibilidad.** Pruebas que se realizan con el fin de mantener el software, realizando, por ejemplo, limpiezas de datos almacenados y que no serán utilizados en el futuro.

- **Pruebas de instalabilidad.** Son pruebas que se realizan en diferentes entornos con el fin de comprobar la dificultad de su instalación.

- **Pruebas de portabilidad.** Pruebas realizadas para observar cuan portable es un software, es decir, cambiarlo de una máquina a otra que posea el mismo entorno.

3.3.2. Las pruebas de caja blanca y pruebas de caja negra

Las pruebas de caja blanca son aquellas que se centran en los procedimientos del software por lo que su diseño está fuertemente ligado al código fuente. Para ello, se escogen distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y garantizar que se devuelven los valores de salida adecuados.

Al estar basadas en una implementación concreta, si esta se modifica, por regla general las pruebas también deberán rediseñarse.

Aunque las pruebas de caja blanca son aplicables a varios niveles, habitualmente se aplican a las unidades de software, en el que se comprobará todos los flujos de ejecución dentro de cada unidad y entre unidades.

Las pruebas de caja negra, por otro lado, son aquellas pruebas en las que se el software se utiliza de manera aislada y sin conocer su código fuente, solo conociendo sus entradas y las salidas que se producen, sin tener en cuenta su funcionamiento interno.

En otras palabras, en las pruebas de caja negra interesa su forma de interactuar con el medio que le rodea, entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace. Por tanto, en estas pruebas deben estar bien definidas sus entradas y salidas. En cambio, no se precisa definir ni conocer los detalles internos de su funcionamiento.

3.3.3. La prueba de mutaciones

La prueba de mutaciones es una técnica de caja blanca que consiste en la introducción de pequeños fallos en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador. Estos fallos son inyectados en base a reglas de transformación predefinidas que simulan fallos de programación habituales, que son denominados *operadores de mutación.*

Para conocer la historia de la prueba de mutaciones debemos remontarnos a 1971, donde un estudiante llamado Richard Lipton dio esta idea en uno de sus artículos [12], pero encontró muchos problemas relacionados con su viabilidad de puesta en marcha en aplicaciones prácticas. Posteriormente, a finales de los años 70, se publicaron los principales artículos sobre este tema con Hamlet en 1977 [13] y con DeMillo en 1978 [14], donde introdujeron formalmente la mutación como una técnica de prueba en sus artículos. Recientemente los avances en la investigación de este tema han acercado a la realidad un sistema práctico de prueba de mutaciones.

En este campo, es denominado *mutante* al código fuente completo, con la inserción de una mutación mediante un operador de mutación específico. A modo de ejemplo se puede considerar el siguiente mutante, siendo el código fuente idéntico al original con el cambio *i=i+1*, cuando en el programa original se contemplaba como *i=i-1*. Una vez obtenido el mutante, este es ejecutado sobre el conjunto de casos de prueba realizados por el desarrollador, pudiendo obtener diferentes salidas al programa original, las cuales pueden ser observadas en la Figura 2.

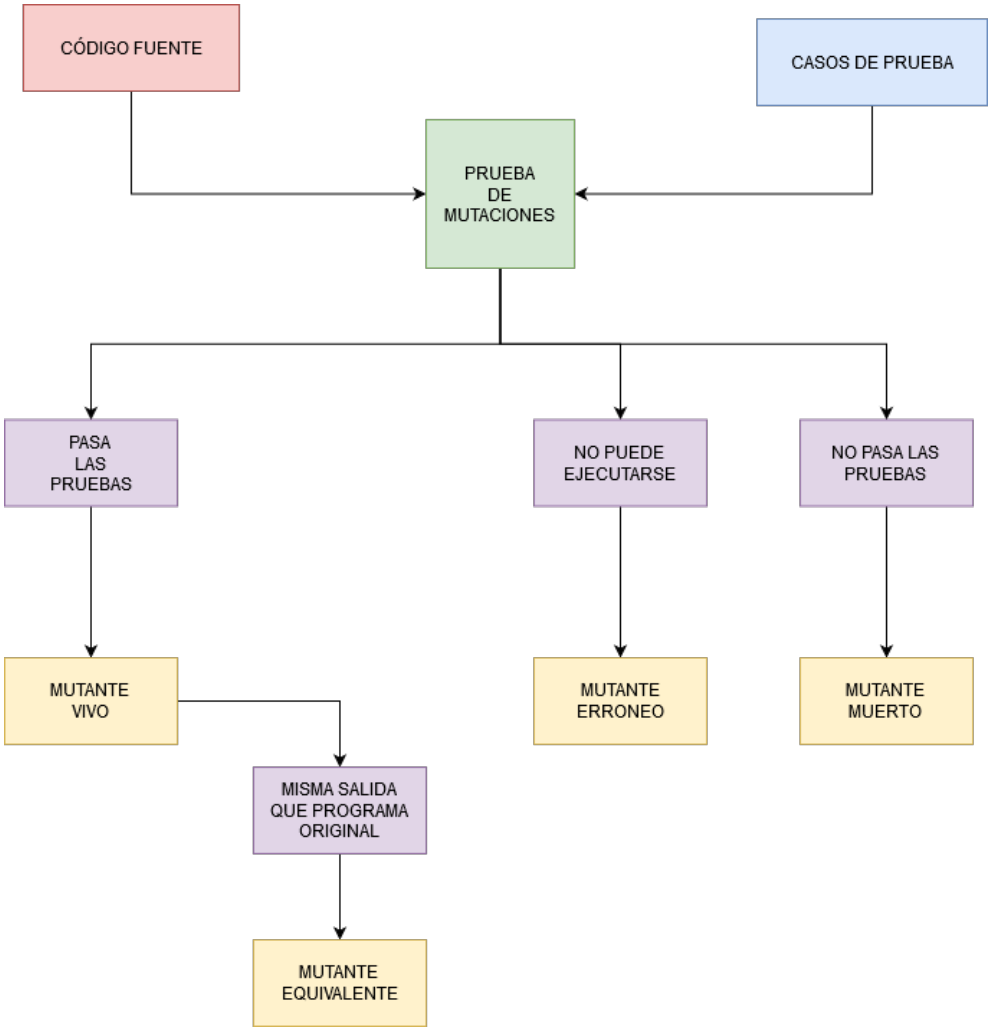

Figura 2: Salidas en la prueba de mutaciones.

Tal y como se ha podido observar, la prueba de mutaciones tiene como entrada el código fuente de un programa y el conjunto de casos de prueba, pudiendo aportar diferentes salidas.

- **Mutante vivo**. El mutante que pasa las pruebas aportadas se considera mutante vivo y significa que las pruebas deben ser mejoradas.
- **Mutante equivalente**. El mutante que da la misma salida que el programa original se considera mutante equivalente.
- **Mutante erróneo**. Aquel mutante que no puede ejecutarse se denomina mutante erróneo.
- **Mutante muerto**. Aquellos mutantes que no pasan las pruebas realizadas por el desarrollador mueren, y tiene como significado que las pruebas realizadas son efectivas con ese mutante.

Un ejemplo de mutante puede ser aquel cuyo operador de mutación es el intercambio de un operador relacional. Por ejemplo, dado un programa P que contiene una instrucción z<10, pueden ser generados diferentes mutantes entre los que se encontrará, por ejemplo, x>10.

Por otro lado, encontramos la forma de medir la calidad de un conjunto de casos de prueba mediante la denominada *puntuación de mutación* (mutation score). Esta puntuación nos indica el porcentaje de mutantes muertos frente al de mutantes no equivalentes. Este porcentaje se calcula de la siguiente forma:

$$PM\ (P,\ C) = \frac{MM}{TM - ME} \cdot 100$$

Siendo:
***PM**: Puntuación de mutación*
***P**: Programa testeado*
***C**: Conjunto de casos de prueba*
***MM**: Mutantes muertos*
***TM**: Total de mutantes*
***ME**: Mutantes equivalentes*

3.3.4. Operadores de mutación para C++11 y C++14

Dado el creciente uso en la industria de C++11 y C++14 con las mejoras que se han ido incluyendo en cada una de las nuevas versiones del lenguaje, en el año 2018, los autores Ali Parsai, Serge Demeyer, y Seph De Busser realizan un artículo para presentar en la *International Conference on Testing Software and Systems*, o ICTSS, su trabajo sobre la creación de operadores de mutación para dichas versiones del conocido lenguaje.

Estos autores realizan el artículo [15], donde dan a conocer una serie de operadores de mutación que han sido probados de forma manual. Dado que estos no han sido probados por una herramienta, se decide comenzar la implementación de dichos operadores para la herramienta MuCPP con el fin de mejorar dichos operadores y realizar diferentes estudios respecto a estos.

Los operadores implementados serán presentados en el presente proyecto como nuevos operadores en la herramienta MuCPP, concretamente en el capítulo 5. Tras la creación de estos operadores se realiza de forma exhaustiva una comparación entre los operadores desarrollados en el artículo y los desarrollados en el presente proyecto con el fin de observar las mejoras implementadas y los resultados obtenidos.

3.4. El proyecto LLVM

3.4.1. LLVM

LLVM [16] es un proyecto que tiene como finalidad la creación de nuevos compiladores optimizados para cualquier lenguaje de programación. Para ello, LLVM suministra la infraestructura necesaria para el desarrollo de compiladores, actuando como backend al tomar el código intermedio que los distintos frontends generan. Se trata de un proyecto de código abierto que engloba un gran número de subproyectos para la creación de nuevos frontends que trabajen sobre LLVM, muchos de los cuales son empleados en otros proyectos, tanto comerciales como de código abierto, así como en el ámbito académico [17].

Para comprender el origen de Clang [**?**], es necesario conocer la existencia del proyecto LLVM, el cual, comenzó como una investigación en la Universidad de Illinois con el objetivo de proporcionar una estrategia de compilación basada en la SSA capaz de soportar tanto la compilación estática como la dinámica [16].

SSA, abreviación de asignación estática única o *static single assignment*, es una propiedad de la representación intermedia de los programas o IR diseñadas para permitir diversas optimizaciones, en la que se requiere que cada variable se asigne únicamente una vez y sea definida antes de usarse, permitiendo al compilador hacer una asignación de registro más eficiente.

El subproyecto de LLVM que se contempla en el presente Proyecto tiene el nombre de Clang, un compilador de código abierto, que proporciona el acceso al AST o Arbol de Sintaxis Abstracta, mediante el cual podremos conseguir el objetivo de la prueba de mutaciones, la inserción de pequeñas modificaciones en el código fuente.

3.4.2. Clang

Clang es un compilador *frontend* de codigo abierto para los lenguajes de programación C, C++ y Objective-C, el cual usa LLVM en su *backend* y cuyo objetivo es una compilación rápida y proporcionar información eficiente sobre errores y advertencias.

Tal y como se mencionó en el apartado anterior, LLVM proporciona la forma intermedia del proceso de compilación mediante un AST, el cual contiene la estructura completa del código fuente que se desea analizar, mediante ramas del árbol, facilitando así el acceso a partes del código. Además, Clang mantiene el código escrito por el desarrollador, cosa que no ocurre en otros compiladores como GCC, los cuales simplifican el código. Esto es beneficioso a la hora de crear mutantes, dado que la intención es conservar el mismo código escrito por el programador e insertarle diferentes modificaciones mediante los diferentes operadores de mutación.

Mediante la combinación de LLVM y Clang se obtiene un importante conjunto de herramientas que permiten el desarrollo de nuevas herramientas como es el caso de MuCPP [18], del cual hablaremos posteriormente.

3.4.3. El árbol de sintaxis abstracta o AST en Clang

Un árbol de sintaxis abstracta o AST es una representación estructurada de un determinado código tras sus correspondientes análisis léxicos, sintáctico y semántico. Las expresiones del código están expresadas mediante la combinación de diferentes ramas del árbol, lo cual nos aporta una visión más clara de la estructura del código que nos facilita la búsqueda de aquellos nodos que cumplen los criterios de los diferentes operadores de mutación.

El AST generado por Clang posee un formato parecido a XML, lo cual lo hace más comprensible incluso por aquellos que aún no están familiarizados por la forma en la que un compilador funciona internamente. Para cada token o nodo, Clang guarda información de en que parte del código fue escrito o donde fue expandido si se trata de una macro.

El AST de Clang es posible extraerlo para poder observar su forma y analizar el código de un fichero de código fuente dado. Esto es posible mediante la siguiente orden ejecutada en una terminal sobre el fichero *ejemplo.cpp*:

```
clang++ -Xclang -ast-dump -fsyntax-only ejemplo.cpp
```

A continuación se expresa un ejemplo del funcionamiento de este AST, haciendo uso de un programa que contiene una única sentencia que es: *a=1+2+3;*. De esta manera, observamos el siguiente AST:

```
-CompoundStmt 0x55b9ea06c1e0 <col:39, line:6:1>
    |-DeclStmt 0x55b9ea06c190 <line:4:2, col:13>
    | `-VarDecl 0x55b9ea06c080 <col:2, col:12> col:6 a 'int' cinit
    |   `-BinaryOperator 0x55b9ea06c168 <col:8, col:12> 'int' '+'
    |     |-BinaryOperator 0x55b9ea06c120 <col:8, col:10> 'int' '+'
    |     |  |-IntegerLiteral 0x55b9ea06c0e0 <col:8> 'int' 1
    |     |  `-IntegerLiteral 0x55b9ea06c100 <col:10> 'int' 2
    |     `-IntegerLiteral 0x55b9ea06c148 <col:12> 'int' 3
```

Tal y como podemos observar, el nivel de sangría o identación que se obtiene en cada una de las sentencias indica la profundidad de un elemento del árbol.

Como se puede ver, en primer lugar se realizan las diferentes operaciones binarias representadas por la clase *BinaryOperator* en la API de Clang, en este caso, primero se realizará la operación *1+2* y posteriormente se le suma el *3*, dado que en primer lugar se realizan las operaciones más internas y posteriormente las más externas. Una vez realizado este cálculo, se declara la variable *a* como entera y se le asigna el valor calculado anteriormente.

Destacar también que estas sentencias son únicamente una parte del árbol generado, siendo la única que hace referencia a la sentencia que nos interesa en este caso, *a=1+2+3;*.

A continuación se puede observar el árbol generado de una forma más visual a la anterior:

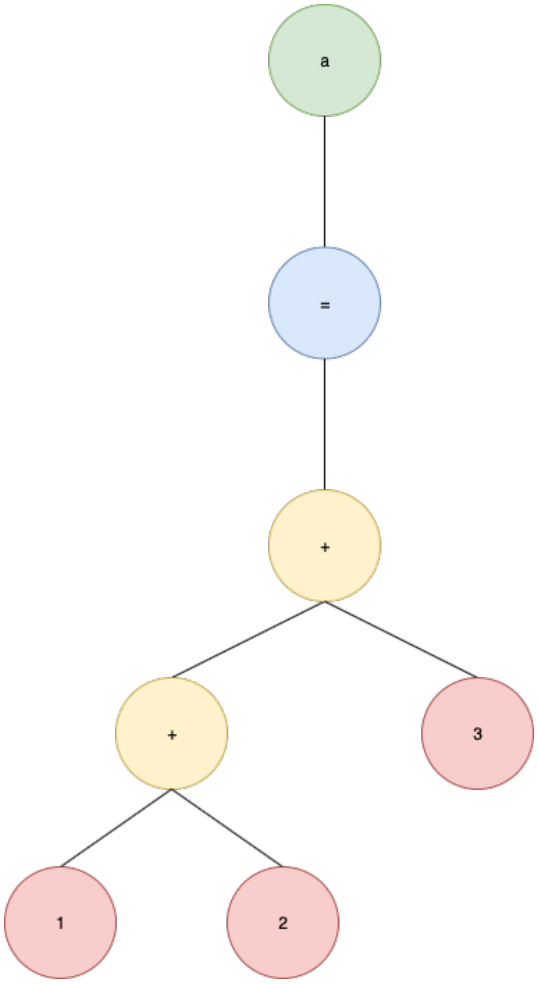

Figura 3: Árbol ejemplo AST.

3.5. Estado del arte

La prueba de mutaciones en sus primeros años de desarrollo se basó en lenguajes estructurados, dado que en esa época aún se encontraban en desarrollo los lenguajes orientados a objetos. Por ello, podemos encontrar herramientas para lenguajes como Ada, Fortran o C.

En la actualidad existen múltiples herramientas para lenguajes como Java o Python, pero en cambio, son mínimas las creadas para el lenguaje C++ y aquellas que se han encontrado son privativas y aportan poca o ninguna información. De hecho, ni siquiera se documentan los operadores de mutación que se pueden encontrar en su interior, por lo que no son de utilidad para el presente proyecto.

En el caso de los sistemas de mutaciones para el lenguaje que más nos importa, C++, las herramientas comerciales existentes incluyen la prueba de mutaciones dentro de un conjunto de técnicas de prueba. Es decir, no se centran en la prueba de mutaciones y además no cubren las mutaciones a nivel de clase sino solo algunas operaciones de carácter

estándar, utilizando la técnica en de manera selectiva. Las herramientas que podemos encontrar son:

- **Certitude Functional Qualification System** (2006) [19]. Esta herramienta combina prueba de mutaciones y análisis estático, calificando un programa funcionalmente y revelando fallos que de otro modo no podrían detectarse. Aunque este producto también se ha utilizado para el análisis de sistemas de software, ahora se dirige a la industria de la microelectrónica.

- **Insure++** (1998) [20]. Esta herramienta utiliza la prueba de mutaciones como una técnica más para mejorar la calidad del software, enfocándose especialmente en problemas de memoria, pero solo realiza algunas mutaciones estándar [21]. Su enfoque es diferente al de la prueba clásica de mutaciones porque solo crea mutantes funcionalmente equivalentes, que se espera que pasen las pruebas en lugar de fallarlas. Por lo tanto, se detecta un error en el programa original cuando se mata un mutante, revelando las ambigüedades que podrían existir. Los usuarios pueden elegir la cantidad de mutantes a generar y aplicar la prueba de mutaciones a una sola función o a un proyecto completo.

- **PlexTest** (2005) [22]. Tal y como se menciona en su página web, este software implementa una prueba de mutaciones altamente selectiva para evitar la generación de mutantes equivalentes. Al seleccionar esta opción, la herramienta solo realiza la mutación de eliminación, eliminando sentencias y subexpresiones. Esta herramienta incorpora algunas otras características para mejorar el rendimiento, como la combinación con un sistema de control de revisión para determinar el código editado recientemente y probar ese código de forma selectiva.

En cuanto a los sistemas de código abierto, CCMutator [23] es un generador de mutaciones para construcciones de concurrencia en C o C ++ recientemente desarrollado. Esta herramienta implementa un conjunto de operadores específicamente diseñados para mutar aplicaciones de subprocesos múltiples.

Dado todo lo anterior, podemos determinar que no existen herramientas de prueba de mutaciones de libre uso que se conozcan, exceptuando la herramienta que se actualiza en el presente proyecto, MuCPP, por lo que la hace de gran interés para su uso en la investigación de este tipo de pruebas. Destacar también la gran cantidad de herramientas para otros lenguajes con poco uso en la industria, como Java, de la que se han desarrollado cinco herramientas, aunque existen otras muchas como Testooj (2007) [24] o Jumble (2001) [25], entre otras.

Capítulo 4

Los operadores de mutación

En el presente capítulo se explicarán las diferentes clases de operadores de mutación que existen y aquellos que podemos encontrar en la herramienta MuCPP, comenzando por los incluidos anteriormente y los incluidos en la herramienta en los últimos años.

4.1. Tipos de operadores de mutación

La herramienta de prueba de mutaciones denominada MuCPP es creada en el seno de la Universidad de Cádiz, concretamente en su Grupo de Investigación UCASE de Ingeniería del Software [1]. Esta herramienta, en un principio, generaba solamente mutantes tradicionales, pasando posteriormente a generar mutantes a nivel de clases mediante diferentes mejoras. MuCPP tiene un sencillo funcionamiento, el cual es explicado en el Apéndice A de este documento, además de contar con una gran cantidad de operadores que le hacen ser un gran competidor frente a otras herramientas para el mismo u otros lenguajes de programación. En la herramienta de prueba de mutaciones MuCPP encontramos todos los tipos de operadores expresados a continuación, tal y como expresa P. Delgado et al. en su artículo [4].

Los operadores de mutación son reglas bien definidas sobre estructuras sintácticas para realizar determinados cambios en el código fuente de un software. Estos operadores pueden ser de diversos tipos, entre los que se encuentran los operadores tradicionales y los operadores de clase, este último contendría diversos subtipos.

- **Operadores de clase**. Operadores que realizan sus acciones sobre clases. Estos operadores pueden ser subdivididos en diferentes tipos, los cuales, son explicados a continuación.

  De herencia. Se centran en todo lo relacionado con la herencia, desde cambiar la llamada a métodos entre clases herederas a eliminar métodos de clases herederas.

  De polimorfismo y enlace dinámico. En esta clase se centran en la polimorfia, y todo lo relacionado con esta.

  De sobrecarga. Se centran en la sobrecarga de métodos.

  De reemplazo. Donde se realizan realizan reemplazos en miembros y objetos de las diferentes clases.

  De manejo de excepciones. Esta clase de operadores se centran en las excepciones, haciendo modificaciones en los manejadores de estas.

Miscelánea. En esta clase de operadores se asignan todos aquellos que no se han asignado en un tipo anterior.

- **Tradicionales**. Operadores que realizan sus acciones sobre métodos concretos o líneas de código fuente concretas. Son denominados como tradicionales ya que fueron los operadores de mutación creados en los comienzos de la técnica y porque abordan elementos que suelen ser comunes a la mayor parte de lenguajes de programación.

4.2. Operadores tradicionales

Bloque	Operador	Descripción
Tradicionales	ARB	Operador de reemplazo aritmético (+, -, *, /, %)
	ARU	Operador de reemplazo de operadores unarios (+, -)
	ARS	Operador de reemplazo de operaciones aritméticas con atajos (++, −−)
	AIU	Operador de inserción de operadores unarios (-)
	AIS	Operador de inserción de atajos (++, −−)
	ADS	Operador de eliminación de atajos (++, −−)
	ROR	Operador de reemplazo de operadores relacionales(<, <=, >, >=, ==, !=)
	COR	Operador de reemplazo de operadores condicionales (&&, ||)
	COD	Operador de eliminación de operadores condicionales (!)
	COI	Operador de inserción de operadores condicionales (!)
	LOR	Operador de reemplazo de operadores lógicos (|, ^)
	ASR	Operador de reemplazo atajos con asignación (%)

Tabla 2: Tabla operadores tradicionales en MuCPP.

Tal y como podemos observar en la tabla 2 existen doce operadores tradicionales en MuCPP, entre los que se encuentran:

[ARB] Operador de reemplazo aritmético

El operador ARB u operador de reemplazo aritmético realiza cambios sintácticos sobre los diferentes operadores aritméticos conocidos cambiándolos por otros. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP			
op1 + op2	op1 − op2	op1 * op2	op1 / op2	op1 % op2
op1 − op2	op1 + op2	op1 * op2	op1 / op2	op1 % op2
op1 * op2	op1 + op2	op1 − op2	op1 / op2	op1 % op2
op1 / op2	op1 + op2	op1 − op2	op1 * op2	op1 % op2
op1 % op2	op1 + op2	op1 − op2	op1 * op2	op1 / op2

Tabla 3: Tabla operador ARB en MuCPP.

[ARU] Operador de reemplazo de operadores unarios

El operador ARU u operador de reemplazo de operadores unarios realiza cambios sintácticos sobre los dos operadores unarios conocidos cambiándo uno por otro, siendo para el operador unario + modificado por - y viceversa. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**				
+op	−op	−−op	op−−	op++	++op
−op	+op	−−op	op−−	op++	++op

Tabla 4: Tabla operador ARU en MuCPP.

[ARS] Operador de reemplazo de operaciones aritméticas con atajos

El operador ARS u operador de reemplazo de operaciones aritméticas con atajos realiza cambios sintácticos sobre los operadores aritméticos con atajos conocidos, cambiando unos por otros. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código Inicial	**Mutantes generados por MuCPP**		
op++	op−−	++op	−−op
++op	−−op	op++	op++
op−−	op++	−−op	op++
−−op	++op	op−−	op++

Tabla 5: Tabla operador ARS en MuCPP.

[AIU] Operador de inserción de operadores unarios

El operador AIU u operador de inserción de operadores unarios, como su nombre indica, inserta cambios sintácticos sobre los operandos encontrados, insertando delante de este el operador unario - u modificando dicho operando por 0. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**	
op	−op	0

Tabla 6: Tabla operador AIU en MuCPP.

[AIS] Operador de inserción de atajos

El operador AIS u operador de inserción de atajos, como su nombre indica, inserta cambios sintácticos sobre los operandos encontrados, insertando alrededor de este diferentes operaciones aritméticas con atajos. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**			
op	−−op	op−−	op++	++op

Tabla 7: Tabla operador AIS en MuCPP.

[ADS] Operador de eliminación de atajos

El operador ADS u operador de eliminación de atajos, como su nombre indica, elimina los operadores unarios ++ y −− encontrados tanto delante como detrás de los diferentes operandos. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**
++op	op
−−op	op
op++	op
op−−	op

Tabla 8: Tabla operador ADS en MuCPP.

[ROR] Operador de reemplazo de operadores relacionales

El operador ROR u operador de reemplazo de operadores relacionales realiza cambios sintácticos sobre los operadores relacionales conocidos, cambiando unos por otros. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**						
op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 >= op2	op1 >op2	op1 <op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 <op2	op1 >op2	op1 >= op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 <= op2	op1 >op2	op1 >= op2	op1 <op2	op1 == op2	op1 != op2	T	F
op1 == op2	op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 != op2	T	F
op1 != op2	op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 == op2	T	F

Tabla 9: Tabla operador ROR en MuCPP.

[COR] Operador de reemplazo de operadores condicionales

El operador COR u operador de reemplazo de operadores condicionales realiza cambios sintácticos sobre los operadores condicionales conocidos, cambiando unos por otros, además de establecer la condición completa como falsa o verdadera. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	**Mutantes generados por MuCPP**							
op1 && op2	op1 || op2	op1 & op2	op1 | op2	op1 ^ op2	op1 == op2	op1 != op2	T	F
op1 || op2	op1 && op2	op1 & op2	op1 | op2	op1 ^ op2	op1 == op2	op1 != op2	T	F

Tabla 10: Tabla operador COR en MuCPP.

[COD] Operador de eliminación de operadores condicionales

El operador COD u operador de eliminación de operadores condicionales, como su nombre indica, elimina los operadores condicionales encontrados junto a un operando. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	Mutantes generados por MuCPP
!op	op

Tabla 11: Tabla operador COD en MuCPP.

[COI] Operador de inserción de operadores condicionales

El operador COI u operador de inserción de operadores condicionales, como su nombre indica, inserta cambios sintácticos sobre los operandos encontrados, insertando alrededor de este un operador condicional. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	Mutantes generados por MuCPP
op	!op

Tabla 12: Tabla operador COI en MuCPP.

[LOR] Operador de reemplazo de operadores lógicos

El operador LOR u operador de reemplazo de operadores lógicos realiza cambios sintácticos sobre los operadores lógicos encontrados, modificando un operador por los diferentes operadores lógicos que podemos encontrar en C++. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	Mutantes generados por MuCPP			
op1 & op2	op1 && op2	op1 || op2	op1 | op2	op1 ^ op2
op1 | op2	op1 && op2	op1 || op2	op1 & op2	op1 ^ op2
op1 ^ op2	op1 && op2	op1 || op2	op1 & op2	op1 | op2

Tabla 13: Tabla operador LOR en MuCPP.

[ASR] Operador de reemplazo de atajos con asignación

El operador ASR u operador de reemplazo de atajos con asignación realiza cambios sintácticos sobre los atajos con asignación encontrados, modificando un atajo por los diferentes atajos con asignación que se pueden encontrar en C++. En la tabla siguiente se puede observar sus funciones de una forma más sencilla.

Código inicial	Mutantes generados por MuCPP			
op1 -= op2	op1 += op2	op1 *= op2	op1 /= op2	op1 %= op2
op1 += op2	op1 -= op2	op1 *= op2	op1 /= op2	op1 %= op2
op1 *= op2	op1 += op2	op1 -= op2	op1 /= op2	op1 %= op2
op1 /= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 %= op2
op1 %= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 /= op2

Tabla 14: Tabla operador ASR en MuCPP.

4.3. Los operadores de herencia en MuCPP

Bloque	Operador	Descripción
De herencia	IHD	Borrado de variable oculta
	IHI	Inserción de variable oculta
	ISD	Borrado de referencia a clase padre
	ISI	Inserción de referencia a clase padre
	IOD	Borrado de método sobrescrito
	IOP	Cambio de posición de la llamada a método sobrescrito
	IOR	Renombrado de método sobrescrito
	IPC	Borrado de llamada explícita al constructor de la clase padre
	IMR	Reemplazo de herencia múltiple

Tabla 15: Tabla operadores de herencia en MuCPP.

Tal y como podemos observar en la tabla 15 existen nueve operadores relacionados con la herencia en MuCPP, entre los que se encuentran:

[IHD] Operador de borrado de variable oculta

El operador IHD u operador de borrado de variable oculta elimina la definición de una variable oculta o sobrescrita por un método hijo, realizando así las referencias a la clase padre en lugar de a la clase hijo. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class Parent{
  int a;
  ...
}
class Child: public Parent{
  int a;
  ...
}
	class Parent{
	  int a;
	  ...
	}
	class Child: public Parent{
	  //int a;
	  ...
	}

Tabla 16: Tabla operador IHD en MuCPP.

[IHI] Operador de inserción de variable oculta

El operador IHI u operador de inserción de variable oculta inserta la definición de una variable oculta, realizando así las referencias a la clase hijo en lugar de a la clase padre. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class Parent{
  int a;
  ...
}
class Child: public Parent{
  ...
}
	class Parent{
	  int a;
	  ...
	}
	class Child: public Parent{
	  int a;
	  ...
	}

Tabla 17: Tabla operador IHI en MuCPP.

[ISD] Operador de borrado de referencia a clase padre

El operador ISD u operador de borrado de referencia a clase padre elimina referencias a la clase padre desde una clase hijo. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class Child: public Parent{
  int metodo(){
  return Parent::a;
  }
  ...
}
	class Child: public Parent{
	  int metodo(){
	  return Parent::a;
	  }
	  ...
	}

Tabla 18: Tabla operador ISD en MuCPP.

[ISI] Operador de inserción de referencia a clase padre

El operador ISI u operador de inserción de referencia a clase padre inserta referencias a la clase padre desde una clase hijo. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class Child: public Parent{
  int metodo(){
  return a;
  }
  ...
}
	class Child: public Parent{
	  int metodo(){
	  return Parent::a;
	  }
	  ...
	}

Tabla 19: Tabla operador ISI en MuCPP.

[IOD] Borrado de método sobrescrito

El operador IOD u operador de borrado de método sobrescrito elimina métodos de la clase padre que han sido sobrescritos en la clase hijo. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class Child: public Parent {
  name(){..}
}
	class Child: public Parent {
	  //name(){..}
	}

Tabla 20: Tabla operador IOD en MuCPP.

[IOP] Operador de cambio de posición de la llamada al método sobrescrito

El operador IOP u operador de cambio de posición de la llamada al método sobrescrito modifica la posición en la que se llama a un método de la clase padre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

class Parent {
  void name(){..}
}
class Child: public Parent {
  void name(){
  Parent::name();
  int a;
  }
}
	class Parent {
	  void name(){..}
	}
	class Child: public Parent {
	  void name(){
	  int a;
	  Parent::name();
	  }
	}

Tabla 21: Tabla operador IOP en MuCPP.

[IOR] Operador de renombrado de método sobrescrito

El operador IOR u operador de renombrado de método sobrescrito renombra los métodos sobrescritos en la clase padre, haciendo que en la clase hijo no se sobrescriba el método de la clase padre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

class Parent {
  void name(){..}
}
class Child: public Parent {
  void name(){}
}
	class Parent {
	  void name'(){..}
	}
	class Child: public Parent {
	  void name(){}
	}

Tabla 22: Tabla operador IOR en MuCPP.

[IPC] Operador de borrado de llamada explícita al constructor de la clase padre

El operador IPC u operador de borrado de llamada explícita al constructor de la clase padre elimina la llamada explícita por parte de una clase hijo al constructor de una clase padre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

class Child: public Parent {
  Child(){
  Parent();
  ...
  }
}
	class Child: public Parent {
	  Child(){
	  //Parent();
	  ...
	  }
	}

Tabla 23: Tabla operador IPC en MuCPP.

[IMR] Operador de reemplazo de herencia múltiple

El operador IMR u operador de reemplazo de herencia múltiple tiene como cometido la modificación o renombrado de la clase a la que se accede para ejecutar un método. Esto sucede cuando una clase hereda de dos o más clases y estas tienen un método en común al que se desea acceder. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

class Parent1 {
  metodo(){..}
}
 	
class Parent2 {
  metodo(){..}
}
 	
class Child: public Parent1, public Parent2 {
  metodo(){
  Parent1::metodo();
  }
}
	class Parent1 {
	  metodo(){..}
	}
	
	class Parent2 {
	  metodo(){..}
	}
	
	class Child: public Parent1, public Parent2 {
	  metodo(){
	  Parent2::metodo();
	  }
	}

Tabla 24: Tabla operador IMR en MuCPP.

4.4. Operadores de polimorfismo y enlace dinámico

Bloque	Operador	Descripción
Operadores de polimorfismo y enlace dinámico	PVI	Inserción del modificador virtual
	PCD	Eliminación del operador de modelado de tipos
	PCI	Inserción del operador de modelado de tipos
	PCC	Cambio del tipo de modelado
	PMD	Declaración con tipo de clase padre
	PPD	Declaración de una variable parámetro con el tipo de una clase hija
	PNC	Llamada al método new con tipo de clase hija
	PRV	Asignación de una referencia con otro tipo compatible

Tabla 25: Tabla de operadores de polimorfismo y enlace dinámico en MuCPP.

Tal y como podemos observar en la tabla 25 existen ocho operadores relacionados con el polimorfismo y el enlace dinámico en MuCPP, entre los que se encuentran:

[PVI] Operador de inserción del modificador virtual

El operador PVI u operador de inserción del modificador virtual tiene como cometido la inserción de la palabra reservada *virtual* en la declaración de un método para el empleo del polimorfismo. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

class A{
  void C(){..}
}
 	
class B: public A{
  void C(){..}
}
	class A{
	  virtual void C(){..}
	}
	
	class B: public A{
	  void C(){..}
	}

Tabla 26: Tabla operador PVI en MuCPP.

[PCI] Operador de inserción del operador de modelado de tipos

El operador PCI u operador de eliminación del operador de modelado de tipos tiene como cometido la inserción del operador de modelado de tipos mediante el operador *dynamic_cast*. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP
Child c; Parent &p = c; p.method();	Child c; Parent &p = c; (dynamic_cast<Child*>(p)).method();

Tabla 27: Tabla operador PCI en MuCPP.

[PCD] Operador de eliminación del operador de modelado de tipos

El operador PCD u operador de eliminación del operador de modelado de tipos tiene como cometido la eliminación del operador de modelado de tipos *dynamic_cast*. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP
Child c; Parent &p = c; (dynamic_cast<Child*>(p)).method();	Child c; Parent &p = c; p.method();

Tabla 28: Tabla operador PCD en MuCPP.

[PCC] Operador de cambio del tipo de modelado

El operador PCC u operador de cambio del tipo de modelado tiene como cometido la modificación del operador de modelado de tipos *dynamic_cast*, cambiando la clase a la que se convertirá por otra que contenga un método con el mismo nombre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP
(dynamic_castChild*(p)).method();	(dynamic_castParent*(p)).method();

Tabla 29: Tabla operador PCC en MuCPP.

[PMD] Operador de declaración con tipo de clase padre

El operador PMD u operador de declaración con tipo de clase padre tiene como cometido la modificación de una declaración de un tipo hijo a un tipo padre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP
Child A; A = new Child();	Parent A; A = new Child();

Tabla 30: Tabla operador PMD en MuCPP.

[PPD] Operador de declaración de una variable parámetro con el tipo de una clase padre

El operador PPD u operador de declaración de una variable parámetro con el tipo de una clase padre tiene como cometido la modificación de una declaración de una variable de un tipo hijo a un tipo padre que es pasada como parámetro. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP
metodo(Child a);	metodo(Parent a);

Tabla 31: Tabla operador PPD en MuCPP.

[PNC] Operador de llamada al método new con tipo de clase hija

El operador PNC u operador de llamada al método new con tipo de clase hija tiene como cometido la modificación de la llamada al método *new* con el tipo de la clase hija en lugar de la clase padre, la cual estaría inicialmente. En la tabla siguiente se puede observar su función.

Código inicial	**Mutantes generados por MuCPP**
Parent A; A = new Parent();	Parent A; A = new Child();

Tabla 32: Tabla operador PNC en MuCPP.

[PRV] Operador de asignación de una referencia con otro tipo compatible

El operador PRV u operador de asignación de una referencia con otro tipo compatible tiene como cometido modificar la asignación de una referencia mediante otra con un tipo compatible. En la tabla siguiente se puede observar su función.

Código inicial	**Mutantes generados por MuCPP**
Parent A; Child1 B; Child2 C; A=&B;	Parent A; Child1 B; Child2 C; A=&C;

Tabla 33: Tabla operador PRV en MuCPP.

4.5. Operadores de sobrecarga

Bloque	**Operador**	**Descripción**
De sobrecarga de métodos	OMD	Borrado de método sobrecargado
	OMR	Cambio del contenido del método sobrecargado
	OAN	Cambio del número de argumentos
	OAO	Cambio del orden de los argumentos

Tabla 34: Tabla operadores de sobrecarga en MuCPP.

Tal y como podemos observar en la tabla 34 existen cuatro operadores relacionados con la sobrecarga en MuCPP.

[OMD] Operador de borrado de método sobrecargado

El operador OMD u operador de borrado de método sobrecargado tiene como cometido eliminar uno de los métodos sobrecargados con los que cuente una clase. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

void metodo(int b){..}
void metodo(bool b){..}
	//Mutante 1
	//void metodo(int b){..}
	void metodo(bool b){..}
	
	//Mutante 2
	void metodo(int b){..}
	//void metodo(bool b){..}

Tabla 35: Tabla operador OMD en MuCPP.

[OMR] Operador de cambio del contenido del método sobrecargado

El operador OMR u operador de cambio del contenido del método sobrecargado tiene como cometido modificar el contenido uno de los métodos sobrecargados con el contenido de otro método con el mismo nombre. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

void metodo(int b){..}
void metodo(bool b){..}
	//Mutante 1
	void metodo(int b){..}
	void metodo(int b, int c){..}
	
	//Mutante 2
	void metodo(int b){..}
	void metodo(int b, int c){
	this-metodo(b);
	}

Tabla 36: Tabla operador OMR en MuCPP.

[OAN] Operador de cambio del número de argumentos

El operador OAN u operador de cambio del número de argumentos tiene como cometido modificar el número de argumentos pasados en una llamada a una función o método. Análogamente al caso anterior, debe existir la sobrecarga que acepte el cambio en la lista de argumentos. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

s.Push(2, 0.5);	//Mutante 1
	s.Push(2);
	
	//Mutante˜2
	s.Push(0.5);
	
	//Mutante 3
	s.Push();

Tabla 37: Tabla operador OAN en MuCPP.

[OAO] Operador de cambio del orden de los argumentos

El operador OAO u operador de cambio del orden de los argumentos tiene como cometido modificar el orden de los argumentos pasados en una llamada a una función o método. En la tabla siguiente se puede observar su función.

Código inicial	**Mutantes generados por MuCPP**
s.Push(2, 0.5);	s.Push(0.5, 2);

Tabla 38: Tabla operador OAO en MuCPP.

4.6. Operadores de manejo de excepciones

Bloque	**Operador**	**Descripción**
Manejo de excepciones	EHC	Cambio del manejador de la excepción
	EHR	Eliminación del manejador de excepción

Tabla 39: Tabla operadores del manejo de excepciones en MuCPP.

Tal y como podemos observar en la tabla 39 existen dos operadores relacionados con las excepciones en MuCPP, entre los que se encuentran:

[EHC] Operador de cambio del manejador de excepciones

El operador EHC u operador de cambio del manejador de excepciones tiene como cometido crear una excepción en lugar de ser tratada en ese mismo momento, modificando así el tratamiento y propagando la excepción. En la tabla siguiente se puede observar su función.

Código inicial	**Mutantes generados por MuCPP**

try{
  ...
}catch(){
  ...
}
	try{
	  ...
	}catch(){
	  throw;
	}

Tabla 40: Tabla operador EHC en MuCPP.

[EHR] Operador de eliminación del manejador de excepciones

El operador EHR u operador de eliminación del manejador de excepciones tiene como cometido modificar el manejador de excepciones eliminando una de las cláusulas catch, siempre que exista más de una. En la tabla siguiente se puede observar su función.

Código inicial	Mutantes generados por MuCPP

try{
...
}catch(int a){
...
}catch(bool b){
...
};
	//Mutante 1
	try{
	...
	//}catch(int a){
	// ...
	}catch(bool b){
	...
	};
	
	//Mutante 2
	try{
	...
	}catch(int a){
	...
	//}catch(bool b){
	// ...
	};

Tabla 41: Tabla operador EHR en MuCPP.

4.7. Operadores de reemplazo

Bloque	Operador	Descripción
De reemplazo	MCO	Llamada a miembro de otro objeto
	MCI	Llamada a miembro de la clase que se hereda

Tabla 42: Tabla operadores del reemplazo en MuCPP.

Tal y como podemos observar en la tabla 39 existen dos operadores relacionados con las excepciones en MuCPP, entre los que se encuentran:

[MCO] Operador de llamada a miembro de otro objeto

El operador MCO u operador de llamada a miembro de otro objeto realiza cambios sintácticos sobre las llamadas a miembros, modificando el objeto al que se realiza la llamada, llamando así a otro objeto de la misma clase. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP
Coche a; Coche b; a.color();	Coche a; Coche b; b.color();

Tabla 43: Tabla operador MCO en MuCPP.

[MCI] Llamada a miembro de la clase que se hereda

El operador MCI u operador de llamada a miembro de la clase que se hereda realiza cambios sintácticos sobre las llamadas a métodos de tipo *virtual*, es decir, métodos de la clase de la que se hereda y se llamará con un objeto de otra clase pero que hereda de la misma clase padre. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP
Gato a; Perro b; a.color();	Gato a; Perro b; b.color();

Tabla 44: Tabla operador MCI en MuCPP.

4.8. Operadores de miscelánea

Bloque	Operador	Descripción
Miscelánea	CTD	Eliminación de la palabra clave this
	CTI	Inserción de la palabra clave this
	CID	Eliminación de la inicialización de una variable
	CDC	Creación del constructor por defecto
	CDD	Eliminación del destructor de clase
	CCA	Eliminación del constructor de copia y del constructor de asignación

Tabla 45: Tabla operadores de miscelánea en MuCPP.

Tal y como podemos observar en la tabla 45 existen seis operadores de esta clase en MuCPP, entre los que se encuentran:

[CTD] Operador de eliminación de la palabra clave this

El operador CTD u operador de eliminación de la palabra clave this elimina las ocurrencias que existan en el código fuente de la palabra clave *this*. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP
this-op	op

Tabla 46: Tabla operador CTD en MuCPP.

[CTI] Operador de inserción de la palabra clave this

El operador CTI u operador de inserción de la palabra clave this inserta la palabra clave *this*, cambiando así el entorno de las expresiones. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP
op	this->op

Tabla 47: Tabla operador CTI en MuCPP.

[CID] Operador de eliminación de la inicialización de una variable

El operador CID u operador de eliminación de la inicialización de una variable elimina la inicialización de una variable. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

A::A(): a(0)˜{b = 1;}	//Mutante 1
	A::A(): a(0)˜{}
	
	//Mutante 2
	A::A()˜{ = 1;}

Tabla 48: Tabla operador CID en MuCPP.

[CDD] Operador de eliminación del destructor de clase

El operador CDD u operador de eliminación del destructor de clase elimina, como su nombre indica, el destructor de la clase. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class A {
...
˜ A(){...};
}
	class A {
	...
	// ˜ A(){...};
	}

Tabla 49: Tabla operador CDD en MuCPP.

[CDC] Operador de creación del constructor por defecto

El operador CDC u operador de creación del constructor por defecto crea, como su nombre indica, el constructor de la clase por defecto. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class clase {
clase(){
std::cout<< "Constructor creado"<<std::endl;
}
}
	class clase {
	clase(){} // Constructor por defecto
	}

Tabla 50: Tabla operador CDC en MuCPP.

[CCA] Operador de eliminación del constructor de copia y del constructor de asignación

El operador CCA u operador de eliminación del constructor de copia y del constructor de asignación elimina, como su nombre indica, los constructores de copia y de asignación. En la tabla siguiente se puede observar sus funciones.

Código inicial	Mutantes generados por MuCPP

class A {
  ...
  A(const A& copy){...}
  A& operator =(const A& copy){...}
};
	//Mutante 1
	class A {
	  ...
	  //A(const A& copy){...}
	  A& operator =(const A& copy){...}
	};
	
	//Mutante 2
	class A {
	  ...
	  A(const A& copy){...}
	  //A& operator =(const A& copy){...}
	};

Tabla 51: Tabla operador CCA en MuCPP.

4.9. Operadores de mutación incluidos en MuCPP en los últimos años

En los últimos años se han introducido nuevos operadores en la herramienta de pruebas de mutaciones MuCPP, con el fin de mantener la herramienta actualizada a las últimas versiones del conocido lenguaje de programación. Estos operadores son los contenidos en la tabla 52.

Operador	Descripción
SDL	Operador de eliminación de sentencias
ODL	Operador de eliminación de operadores
CSD	Operador de eliminación de la palabra clave *static*
CSI	Operador de inserción de la palabra clave *static*

Tabla 52: Tabla de operadores incluidos en MuCPP en los últimos años.

A continuación se detallarán cada uno de los operadores, su fin y la acción que realizan.

[SOR] Operador de reemplazo de operadores de desplazamiento

El operador SOR u operador de reemplazo de operadores de desplazamiento realiza un cambio del operador de desplazamiento, consiguiendo así incluir los operadores de desplazamiento entre los mutantes de la herramienta, los cuales no habían sido incluidos hasta el momento dado su poco uso. Con esta inclusión se determina su importancia dado que su poco uso puede llevar al programador a confundir su finalidad o su funcionamiento.

Este operador intercambia los operadores de desplazamiento << por >> y viceversa, además del operador de desplazamiento con asignación <<= por >>= y viceversa. En la tabla siguiente se puede observar sus funciones.

Código inicial	**Mutantes generados por MuCPP**
op = op1<<op2	op = op1>>op2
op = op1>>op2	op = op1<<op2
op1 <<= op2	op1 = op2
op1 = op2	op1 = op2

Tabla 53: Tabla operador SOR en MuCPP.

[SDL] Operador de eliminación de sentencias

El operador SDL u operador de eliminación de sentencias elimina todas y cada una de las sentencias ejecutables de un código fuente. Cuando se aplica a estructuras de control que incluyen un bloque de declaraciones (por ejemplo, if, while o for), se elimina todo el bloque, así como cada instrucción interna.

Este operador fue expuesto por Lin Deng, Jeff Offutt y Nan Li en [26], en el que expresan que comenzaron este operador con declaraciones individuales y posteriormente ampliaron a otras estructuras de control. Además de esto se expresa las acciones que se deben realizar sobre todas y cada una de las diferentes estructuras de control más conocidas.

Por un lado podemos observar que se actua sobre los bucles de control while, en los que se crea un mutante poniendo la condición a True, es decir, ejecutando infinitamente el bloque de control y otro eliminando el bloque while al completo. A continuación podemos observar un ejemplo de los mutantes que se crean sobre este bloque:

Código Inicial	**Mutantes en MuCPP**	

...
while(x!= 0){
  x=x+1;
}
...
	...
	while(true){
	  x=x+1;
	}
	...
		
		...
		...

Tabla 54: Tabla operador SDL sobre bloques while.

Por otro lado podemos observar como se actua sobre los bloques for en los que se elimina en primer lugar el bloque completo, posteriormente se crea otro mutante eliminando la condición y, por último, se crea un mutante eliminando el incremento/decremento. A continuación podemos observar en la tabla los diferentes mutantes que se crearían con este operador.

También encontramos los bloques if-else, en los que, en primer lugar, se pondrá la condición a verdadero, haciendo que pase por la parte del verdadero si o si. Por otro lado se pondrá a falsa, haciendo que si existe el else, pase por dicha parte y en caso contrario esto hará que no entre en la condición if, haciendo como si hubiera sido eliminada, por lo que no se borrará el bloque. En caso de que exista el else, también se borrarán ambos bloques, haciendo así que no pase por ninguno de los dos. A continuación puede observar este caso en la siguiente tabla.

Código inicial	Mutantes generados por MuCPP		

...
for(int i=0; i10; i++){
x=x+1;
}
...
	...
	...
		...
		for(int i=0; ; i++){
		x=x+1;
		}
		...
			...
			for(int i=0; i10;){
			x=x+1;
			}
			...

Tabla 55: Tabla operador SDL sobre bloques For en MuCPP.

Código inicial	Mutantes generados por MuCPP		

...
if(x=0){
...
}else{
...
}
...
	...
	if(true){
	...
	}else{
	...
	}
	...
		...
		if(false){
		...
		}else{
		...
		}
		...
			...
			...

Tabla 56: Tabla operador SDL sobre bloques If-Else en MuCPP.

Por otro lado se ocupará de los bloques switch, en los que se eliminarán uno a uno cada uno de los casos que este contenga. Eliminando así el caso completo y no cada una de las sentencias que este contenga.

En quinto lugar se realizan las devoluciones de las funciones, es decir, los *return*, haciendo que se devuelva cero en caso de que nos encontremos que se está devolviendo un *Builtin* y *NULL* en caso contrario, con esto conseguimos que si nos encontramos ante una dirección de memoria se devuelva NULL y en cualquier otro caso se devuelva un cero. En [27] se puede observar que los tipos Builtin son todos los básicos, es decir, int, char, float, bool, etc. Es por ello que se usa esta técnica dado que es considerada más sencilla a la hora de programar el código. A continuación se pueden observar cada uno de los tipos y lo que devuelve MuCPP cuando crea el mutante.

Tipo del return	Mutantes generados por MuCPP
int	return 0;
char	return 0;
bool	return 0;
float	return 0;
double	return 0;
array	return NULL;
puntero	return NULL;
referencia	return NULL;
clase	return NULL;
estructura	return NULL;
union	return NULL;
enumeración	return NULL;
typedef	return NULL;

Tabla 57: Tabla operador SDL sobre returns en MuCPP.

Además si nos encontramos ante un SwitchCase se eliminará la primera sentencia, pero no el "case –:", como si se realizaría si no tuviéramos esta parte.

Y por último se examinan las sentencias individuales, ya sean internas a un bloque o no, ante las cuales se elimina la sentencia al completo. Es por ello, que en cada uno de los bloques anteriores no se ha expresado la eliminación de cada una de las sentencias internas.

[ODL] Operador de eliminación de operadores

El operador ODL u operador de eliminación de operadores elimina cualquier operador existente en C++, ya sea aritmético, relacional, lógico, de bit a bit o de asignación. Cuando es el caso de la eliminación de un operador de asignación (por ejemplo, "a + = 2") simplemente se elimina el operador y se sustituye por una asignación simple ("a = 2"). En el caso del operador binario también se debe eliminar un operando para que la expresión esté correcta para poder ser compilada. Es por ello que al eliminar un operador binario se crean dos mutantes, uno cuando se elimina el operando derecho y otro cuando se elimina el operando izquierdo.

Código inicial	**Mutantes generados por MuCPP**	
op1 + op2	op1	op2
op1 - op2	op1	op2
op1 * op2	op1	op2
op1 / op2	op1	op2
op1 % op2	op1	op2
op1 op2	op1	op2
op1 op2	op1	op2
op1 = op2	op1	op2
op1 = op2	op1	op2
op1 == op2	op1	op2
op1 != op2	op1	op2
op1 op2	op1	op2
op1 op2	op1	op2
op1 || op2	op1	op2
op1 | op2	op1	op2
op1 ^ op2	op1	op2
op1 op2	op1	op2
op1 op2	op1	op2
op1 += op2	op1 = op2	
op1 -= op2	op1 = op2	
op1 *= op2	op1 = op2	
op1 /= op2	op1 = op2	
op1 % op2	op1 = op2	
op1 = op2	op1 = op2	
op1 = op2	op1 = op2	
op1 = op2	op1 = op2	
op1 |= op2	op1 = op2	
op1 ^= op2	op1 = op2	

Tabla 58: Tabla operador ODL en MuCPP.

Tal y como podemos observar en la tabla, se crean dos mutantes en el caso de los operadores binarios con asignación y uno en el caso que contiene asignación.

[CSD] Operador de eliminación de las palabras clave inline static

El operador CSD u operador de eliminación de las palabras clave inline static elimina las palabras claves *inline static* tras la inclusión en c++17 de la posibilidad de inicialización de las variables static en la misma línea mediante la clausula inline. Cabe destacar que este operador solo se utiliza en caso de que la sentencia sea de la forma *inline static tipo var = valor;*.

Código inicial	**Mutantes generados por MuCPP**
inline static int a = 5;	int a = 5;

Tabla 59: Tabla operador CSD en MuCPP.

[CSI] Operador de inserción de las palabras clave inline static

El operador CSI u operador de inserción de las palabras clave inline static inserta las palabras claves *inline static* tras la inclusión en c++17 de la posibilidad de inicialización de las variables static en la misma línea mediante la clausula inline. Cabe destacar que este operador solo se utiliza en caso de que la sentencia sea de la forma *tipo var = valor;*, ya que se desea que la variable esté inicializada en el momento ya que no se podría inicializar posteriormente.

Código inicial	**Mutantes generados por MuCPP**
int a = 5;	inline static int a = 5;

Tabla 60: Tabla operador CSI en MuCPP.

Capítulo 5

Nuevos operadores de mutación para C++11 y C++14 en MuCPP

En la literatura han aparecido diferentes artículos relacionados con la realización de diferentes mutantes en las versiones concretas de C++11 y C++14. Por ello, mediante esta sección se crearán dichos operadores de mutación, incluyendo diversas mejoras que a lo largo de la sección serán explicadas en detalle.

5.1. [FOR] Operador de eliminación de referencia de rango en bucles for

La presente sección se divide en dos apartados bien diferenciados. En primer lugar se explicará la utilidad real del operador, pasando por las salidas que genera y en segundo lugar se explicará la implementación realizada de dicho operador.

5.1.1. Utilidad del operador

El operador de eliminación de referencia (FOR) basada en el rango elimina referencias con la forma for (T& elem : rango) o for (T&& elem : rango), donde T es auto o de un tipo concreto, y elimina el calificador de referencia de la declaración del rango. En la tabla siguiente se puede observar diferentes salidas que proporciona este operador.

Código inicial	**Mutantes en MuCPP**
for(auto& elem : range){. . .}	For(auto elem : range){. . .}
for(auto&& elem : range){. . .}	For(auto elem : range){. . .}

Tabla 61: Tabla de utilidad del operador FOR

El operador FOR se basa en la posibilidad de confusión sobre la semántica de valores por defecto del nuevo rango basado en bucles, mientras que los métodos anteriores de bucles sobre contenedores daban como resultado una semántica de referencia.

Sabiendo esto se hace uso de las estadísticas obtenidas por [15] y sus conclusiones sobre los principales problemas causantes de mutantes inválidos y mutantes equivalentes, creando así un operador que genere el menor número de mutantes no deseados.

5.1.2. Implementación

A continuación se presenta el AST Matcher que usará Clang para encontrar las lineas de código ante las que se podrá actuar con este operador.

```
DeclarationMatcher FOR_Matcher =
    functionDecl(
        forEachDescendant(
            cxxForRangeStmt(
                hasLoopVariable(
                    varDecl(
                        hasType(
                            qualType(
                                references(
                                    qualType(
                                        unless(
                                            isConstQualified()
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            ).bind("FOR")
        )
    );
```

Tal y como se puede observar en el AST, se buscarán bucles for con rango, los cuales serán marcados para ser tratados posteriormente. Estos bucles for con rango deberán poseer variables internas de referencia que no posean el calificativo de constante, los cuales serán pasadas por alto.

Por otro lado, se ha procedido a implementar la función que realizará las acciones del operador de mutación y que podemos observar a continuación.

```
virtual void run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    Rewriter Rewrite;
    Rewrite.setSourceMgr(Context->getSourceManager(), Context->getLangOpts());

    if (const CXXForRangeStmt *FS =
        Result.Nodes.getNodeAs<clang::CXXForRangeStmt>("FOR")){

        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getLocStart());
        if (FullLocation.isValid() &&
```

```
            !Context->getSourceManager().isInSystemHeader(FullLocation)){

            llvm::outs() << "Found declaration at "
                << FullLocation.getSpellingLineNumber() << ":"
                << FullLocation.getSpellingColumnNumber() << "\n";

            SourceLocation sl_begin = FS->getLoopVariable()->getOuterLocStart();
            SourceLocation sl_source = findSymbolAfterLocation(sl_begin, Context, '&');
            SourceRange rt = SourceRange(sl_source, sl_source.getLocWithOffset(1));
            string label = "/*FOR*/";
            Rewrite.ReplaceText(rt, label);
            const RewriteBuffer *RewriteBuf =
                Rewrite.getRewriteBufferFor(FullLocation.getFileID());
            llvm::outs() << std::string(RewriteBuf->begin(), RewriteBuf->end());
        }
    }
}
```

Tal y como se puede observar, el operador obtiene la localización del comienzo del iterador y posteriormente obtiene la localización concreta del símbolo &. Una vez obtenidas esas localizaciones, se sobrescribe el lugar en el que se encuentra el símbolo o par de símbolos.

Cabe destacar que se hace uso de la función interna de MuCPP denominada *findSymbolAfterLocation*, la cual devuelve la posición del símbolo pasado después de la ubicación proporcionada.

Para reducir la cantidad de mutantes equivalentes, se ha tenido en cuenta las recomendaciones indicadas por el autor del artículo y su análisis, dejando de lado aquellos bucles que poseen referencias constantes. En cuanto a los mutantes muertos, no se han podido tener en cuenta, dado que no podemos saber de antemano si el elemento tendrá o no constructor de copia.

Cabe destacar que en el capítulo siguiente se realizará un análisis exhaustivo de este y otros operadores.

5.2. [LMB] Operador de paso por referencia en funciones lambda

La presente sección se divide en dos apartados bien diferenciados. En primer lugar se explicará la utilidad real del operador, pasando por las salidas que genera y en segundo lugar se explicará la implementación realizada de dicho operador.

5.2.1. Utilidad del operador

El operador de paso por referencia en funciones lambda cambia el paso por valor predeterminado a paso por referencia. En la tabla siguiente se muestra un ejemplo de mutante generado a partir de un código original.

Código inicial	Mutante en MuCPP
[=](int z) {return z + 1;};	[&](int z) {return z + 1;};

Tabla 62: Tabla de utilidad del operador LMB

El operador LMB se basa en las advertencias sobre la captura por defecto en la directriz básica F53. Este operador de mutación resulta en un código que lleva a un comportamiento indefinido si la lambda se ejecuta en un contexto no local, porque las referencias a las variables locales no son válidas.

Este operador se basa en un cambio sintáctico menor que puede ser fácilmente pasado por alto. Los mutantes creados por este operador tampoco son fáciles de detectar, porque invocan un comportamiento indefinido que es altamente dependiente de los niveles de optimización del compilador y de las circunstancias en tiempo de ejecución.

Sabiendo esto se hace uso de las estadísticas obtenidas por [15] y sus conclusiones sobre los principales problemas causantes de mutantes inválidos y mutantes equivalentes, creando así un operador que genere el menor número de mutantes no deseados.

5.2.2. Implementación

A continuación se presenta el AST Matcher que usará Clang para encontrar las lineas de código ante las que se podrá actuar con este operador.

```
DeclarationMatcher LMB_Matcher =
functionDecl(
    forEachDescendant(
        lambdaExpr().bind("LMB")
    )
);
```

Tal y como se puede observar en el AST Matcher, se buscarán funciones declaradas que sean expresiones lambda, los cuales serán marcadas para ser tratadas posteriormente por la función encargada de realizar los cambios introducidos por el operador que estamos tratando.

Por otro lado, se ha procedido a implementar la función que realizará las acciones del operador de mutación y que podemos observar a continuación.

```
virtual void run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    Rewriter Rewrite;
    Rewrite.setSourceMgr(Context->getSourceManager(), Context->getLangOpts());

    if (const LambdaExpr *FS = Result.Nodes.getNodeAs<clang::LambdaExpr>("LMB")){
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getLocStart());
        if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)){

            tring label = "/*LMB*/&";

            if(FS->getCaptureDefault() == LCD_ByCopy){
```

```
                SourceRange rt = SourceRange(FS->getCaptureDefaultLoc(),
                    FS->getCaptureDefaultLoc().getLocWithOffset(0));
                Rewrite.ReplaceText(rt, label);
            }else{

for(LambdaExpr::capture_iterator f1 = FS->capture_begin();
    f1 != FS->capture_end(); f1++){
LambdaCapture LC = *f1;
if( LC.getCaptureKind() == LCK_ByCopy &&
    !LC.getCapturedVar()->isInitCapture()) {
        Rewrite.InsertTextBefore(LC.getLocation(), label);
    }
}
}
const RewriteBuffer *RewriteBuf =
    Rewrite.getRewriteBufferFor(FullLocation.getFileID());
llvm::outs() << std::string(RewriteBuf->begin(),
    RewriteBuf->end());
}
}
  }
```

Tal y como se puede observar, una vez obtenida la localización proporcionada por el AST Matcher, la expresión lambda es asignada a un objeto constante de la clase LambdaExpr. Tras esto, se comprueba que las expresiones se están pasando por copia y en caso afirmativo, esta es modificada por el paso por referencia.

Cabe destacar que si se encuentra más de una expresión, estas serán recorridas comprobando si se pasan por copia y en caso afirmativo se modificará por un paso por referencia. Finalmente se reescribe el código con los cambios introducidos por el operador de mutación.

Por último cabe destacar que en [15] solo se hace referencia a aquellas funciones lambda que son del tipo [=](){} y no al resto de funciones lambda. Es por ello que mediante este operador, se ha intentado mejorar sus funciones incluyendo cualquier lista de paso por valor y/o referencia y modificando los pasos por valor por pasos por referencia. Una mejora que sin ninguna duda, tal y como se puede observar en el capítulo siguiente, mejora las estadísticas de este operador y lo hace mucho más competitivo que el original.

5.3. [FWD] Operador de modificación de llamadas a forward por llamadas a move

La presente sección se divide en dos apartados bien diferenciados. En primer lugar se explicará la utilidad real del operador, pasando por las salidas que genera y en segundo lugar se explicará la implementación realizada de dicho operador.

5.3.1. Utilidad del operador

El operador de modificación de forward a move, tal y como su nombre indica, modifica las llamadas a std::forward por llamadas a std::move. En la tabla siguiente se muestra un ejemplo de mutante generado a partir de un código original.

Código inicial	Mutantes en MuCPP

template <class T>
void wrapper (T arg){
foo(std::forwardT(arg));
}
	template <class T>
	void wrapper (T arg){
	foo(std::move(arg));
	}

Tabla 63: Tabla de utilidad del operador FWD

El operador FWD se basa en la sustitución de std::forward por std::move, lo cual tiene un gran potencial debido a que esto puede cambiar el comportamiento del programa. Sabiendo esto se hace uso de las estadísticas obtenidas por [15] y sus conclusiones sobre los principales problemas causantes de mutantes inválidos y mutantes equivalentes, creando así un operador que genere el menor número de mutantes no deseados.

5.3.2. Implementación

A continuación se presenta el AST Matcher que usará Clang para encontrar las lineas de código ante las que se podrá actuar con este operador.

```
DeclarationMatcher FWD_Matcher = functionDecl(
    forEachDescendant(
        callExpr(
            callee(
                functionDecl(
                    hasName("forward")
                )
            )
        ).bind("FWD")
    )
);
```

Tal y como se puede observar en el AST Matcher, se buscarán funciones declaradas que sean llamadas cuyo nombre sea forward. Dichas llamadas serán marcadas para ser tratadas posteriormente por la función encargada de realizar los cambios introducidos por el operador que estamos tratando.

Por otro lado, se ha procedido a implementar la función que realizará las acciones del operador de mutación y que podemos observar a continuación.

```
virtual void run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    Rewriter Rewrite;
    Rewrite.setSourceMgr(Context->getSourceManager(), Context->getLangOpts());

    if (const CallExpr *FS = Result.Nodes.getNodeAs<clang::CallExpr>("FWD")){
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getLocStart());

        if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)){
```

```
            if ( FS->getDirectCallee()->isInStdNamespace() ){

                SourceRange rt = FS->getCallee()->getSourceRange();
                SourceLocation sl_begin = rt.getBegin();
                SourceLocation sl_source =
                    findSymbolAfterLocation(sl_begin, Context, ':');

                if( sl_source == rt.getEnd() )
                    sl_begin = rt.getBegin();
                else
                    sl_begin = sl_source.getLocWithOffset(2);

                sl_source = findSymbolAfterLocation(sl_begin, Context, '<');
                rt = SourceRange(sl_begin, sl_source.getLocWithOffset(-1));
                string label = "move/*FWD*/";
                Rewrite.ReplaceText(rt, label);
                const RewriteBuffer *RewriteBuf =
                    Rewrite.getRewriteBufferFor(FullLocation.getFileID());
            }
        }
    }
}
```

Tal y como se puede observar, una vez obtenida la localización proporcionada por el AST Matcher, la llamada a expresión es asignada a un objeto constante de la clase CallExpr. Tras esto, se comprueba que la expresión se encuentra en el espacio de nombre std. Posteriormente se obtiene la ubicación exacta de la palabra ̈forwardÿ se modifica por la palabra move, ya tenga el espacio de nombre mediante *::* o mediante *using namespace*.

Cabe destacar que se hace uso de la función interna de MuCPP denominada *findSymbolAfterLocation*, la cual devuelve la posición del símbolo pasado después de la ubicación proporcionada.

Mutantes Inválidos: Los mutantes inválidos estaban compuestos por dos grupos: el argumento de plantilla fija y los argumentos de referencia de valores no constantes. El primer grupo reenvía a otra función de plantilla, mientras que el argumento de plantilla se indica explícitamente como se ve en el extracto de código 1.10. Esto hace que el código no se compile cuando se le llama con un valor l no constante. Si se llama con valores constantes o referencias rvalue tendrá el mismo comportamiento en tiempo de ejecución que el original.

Cabe destacar que en el capítulo siguiente se realizará un análisis exhaustivo de este y otros operadores.

5.4. [INI] Operador de llamada al constructor de inicialización con lista

La presente sección se divide en dos apartados bien diferenciados. En primer lugar se explicará la utilidad real del operador, pasando por las salidas que genera y en segundo lugar se explicará la implementación realizada de dicho operador.

5.4.1. Utilidad del operador

El operador de constructor de inicialización con lista comprueba las llamadas realizadas al constructor de tipos con una lista y realiza un cambio desde o hacia una inicialización uniforme en orden para llamar a un constructor diferente. En la tabla siguiente se muestra un ejemplo de mutante generado a partir de un código original.

Código inicial	Mutantes en MuCPP
std::vector<float>v(1.5, 1.0);	std::vector<float>v{1.5, 1.0};
std::vector<float>v{1.5, 1.0};	std::vector<float>v(1.5, 1.0);

Tabla 64: Tabla de utilidad del operador INI

El operador INI se basa en las advertencias sobre la captura por defecto en la directriz básica F53. Este operador de mutación resulta en un código que lleva a un comportamiento indefinido si la lambda se ejecuta en un contexto no local, porque las referencias a las variables locales no son válidas.

Si bien los constructores de listas de inicialización son útiles para definir el contenido de un contenedor, también son posibles fuentes de fallos. Por ejemplo, cuando se usa una inicialización uniforme hay que prestar atención a la sintaxis correcta, ya que al usar {} en lugar de () por error se cambia la semántica de la expresión drásticamente. Un ejemplo destacado de este problema es std::vector¡int¿. La versión no mutada en puede ser vector¡int¿a(1,2) extracto de código que define un vector de un elemento con valor 2, mientras que el vector mutado vector¡int¿a{1, 2} tiene dos elementos: 1 y 2.

Sabiendo esto se hace uso de las estadísticas obtenidas por [15] y sus conclusiones sobre los principales problemas causantes de mutantes inválidos y mutantes equivalentes en este operador, creando así un operador que genere el menor número de mutantes no deseados.

5.4.2. Implementación

A continuación se presenta el AST Matcher que usará Clang para encontrar las lineas de código ante las que se podrá actuar con este operador.

```
DeclarationMatcher INI_Matcher = functionDecl(
    forEachDescendant(
        declStmt(
            hasDescendant(
                varDecl(
                    hasDescendant(
                        constructExpr(
                            hasDeclaration(
                                methodDecl(
                                    ofClass(
                                        recordDecl().bind("INI2RD")
                                    )
                                )
                            )
                        ).bind("INI2CE")
                    )
                )
```

```
            )
        )
    )
);
```

Tal y como se puede observar en el AST Matcher, se buscarán funciones declaradas que seaun un constructor, los cuales se marcarán para ser tratados posteriormente. Estos constructores deberán tener una declaración, la cual será marcada también para ser usada.

Por otro lado, se ha procedido a implementar la función que realizará las acciones del operador de mutación y que podemos observar a continuación.

```
virtual void run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    Rewriter Rewrite;
    Rewrite.setSourceMgr(Context->getSourceManager(), Context->getLangOpts());
    const CXXRecordDecl *FS = 0;
    const CXXConstructExpr *ce = 0;
    if(
        (FS = Result.Nodes.getNodeAs<clang::CXXRecordDecl>("INI2RD"))
        && (ce = Result.Nodes.getNodeAs<clang::CXXConstructExpr>("INI2CE"))
    ){
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(ce->getLocStart());
        if (
            FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)
        ){
            if(ce->isStdInitListInitialization()){
            llvm::outs() << " ENTRO\n";
            SourceLocation s = findSymbolAfterLocation(
                ce->getLocStart(),Context,'{'
            );
            SourceRange r1 = SourceRange(s, s);
            string label = "/*INI*/(";
            Rewrite.ReplaceText(r1, label);
            s = findSymbolAfterLocation(ce->getLocStart(),Context,'}');
            r1 = SourceRange(s, s);
            label = ")";
            Rewrite.ReplaceText(r1, label);
            const RewriteBuffer *RewriteBuf =
                Rewrite.getRewriteBufferFor(FullLocation.getFileID());
                llvm::outs() << std::string(
                    RewriteBuf->begin(), RewriteBuf->end()
                );

        }else{
            for(CXXRecordDecl::ctor_iterator c = FS->ctor_begin();
                c != FS->ctor_end(); c++){
```

```
                unsigned int num_params = (*c)->getNumParams();
                for(unsigned int p = 0; p < num_params; p++){
                    if(
                        (*c)->getParamDecl(p)->getType().getCanonicalType()
                        .getAsString().find("initializer_list")
                        != std::string::npos
                    ){
                        SourceLocation s = findSymbolAfterLocation(
                            ce->getLocStart(),Context,'('
                        );
                        SourceRange r1 = SourceRange(s, s);
                        string label = "/*INI*/{";
                        Rewrite.ReplaceText(r1, label);
                        s = findSymbolAfterLocation(
                            ce->getLocStart(),Context,')'
                        );
                        r1 = SourceRange(s, s);
                        label = "}";
                        Rewrite.ReplaceText(r1, label);
                        const RewriteBuffer *RewriteBuf =
                        Rewrite.getRewriteBufferFor(FullLocation.getFileID());
                        llvm::outs() << std::string(
                            RewriteBuf->begin(), RewriteBuf->end()
                        );
                    }
                }
            }
        }
    }
}
```

Tal y como se puede observar en la funcion anterior, existen dos partes bien diferenciadas, las cuales pasamos a detallar a continuación. Por un lado podemos encontrar, la parte encargada de mutar las llaves por paréntesis. Tal y como podemos ver en primer lugar se obtiene la localización concreta de las llaves y posteriormente son sustituidos por paréntesis. Por otro lado podemos encontrar la parte encargada de mutar los paréntesis por llaves. Tal y como podemos ver en primer lugar se obtiene la localización concreta de los paréntesis y posteriormente son sustituidos por llaves.

Cabe destacar que en la versión inicial expuesta en [15] no se incluye la mutación de llaves a paréntesis, lo cual se considera una funcionalidad con sentido. Es por ello que mediante este operador, se ha intentado mejorar sus funciones incluyendo esta funcionalidad. Una mejora que sin ninguna duda, tal y como se puede observar en capítulo siguiente, mejora las estadísticas de este operador y lo hace mucho más competitivo que el original.

Capítulo 6

Pruebas sobre nuevos operadores

En este capítulo podemos encontrar algunas de las pruebas realizadas a todos y cada uno de los operadores de nueva inclusión en la herramienta MuCPP. Estas pruebas se componen en primer lugar del código de entrada a MuCPP y las salidas (mutantes) que nos devuelve. Los operadores a los que se han realizado las pruebas pueden observarse en la tabla 65.

Operador	Descripción
FOR	Operador de eliminación de referencia de rango en bucles for
LMB	Operador de paso por referencia en funciones lambda
FWD	Operador de modificación de llamadas a forward por llamadas a move
INI	Operador de llamada al constructor de inicialización con lista

Tabla 65: Tabla de operadores incluidos en MuCPP en el presente proyecto.

6.1. Pruebas sobre el operador FOR

La entrada para este operador se compone de una función main que contempla diferentes circunstancias para la sentencia FOR, comprobando así la utilidad del operador correspondiente.

```
1   #include <iostream>
2   #include <vector>
3
4   int main() {
5       std::vector<int> v = {0, 1, 2, 3, 4, 5};
6
7       for (const int& i : v)
8           std::cout << i << ' ';
9       std::cout << '\n';
10
11      for (auto i : v)
12          std::cout << i << ' ';
13      std::cout << '\n';
```

```
14
15          for (auto& i : v)
16              std::cout << i << ' ';
17          std::cout << '\n';
18
19          const auto& cv = v;
20
21          for (auto&& i : cv)
22              std::cout << i << ' ';
23          std::cout << '\n';
24
25          for (auto&& i : v)
26              std::cout << i << ' ';
27          std::cout << '\n';
28
29          for (int n : {0, 1, 2, 3, 4, 5})
30              std::cout << n << ' ';
31          std::cout << '\n';
32
33          int a[] = {0, 1, 2, 3, 4, 5};
34          for (int n : a)
35              std::cout << n << ' ';
36          std::cout << '\n';
37
38          for (int n : a)
39              std::cout << 1 << ' ';
40          std::cout << '\n';
41
42      }
```

Este código devuelve un total de dos mutantes. Destacar también que la herramienta tiene asignado el identificador 48 para el operador FOR, el cual estamos probando.

```
The mutant 'm48_1_1_Pruebas_FOR' has been created.
The mutant 'm48_2_1_Pruebas_FOR' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente.

1. El mutante m48_1_1_Pruebas_FOR contiene la siguiente mutación en la línea 15: ***for (auto i : v)***.
2. El mutante m48_2_1_Pruebas_FOR contiene la siguiente mutación en la línea 25: ***for (auto i : v)***.

6.2. Pruebas sobre el operador LMB

La entrada para este operador se compone de una función main con diferentes llamadas a funciones lambda, en las cuales se debe aplicar el operador LMB, el cual estamos probando.

```
1   #include <iostream>
2
```

```
3   int main()
4   {
5       int a, b;
6       [=](int x) { return x > a+b;};
7       [a,b](int x) { return x > a+b;};
8       [&,b](int x) { return x > a+b;};
9       [a,c = b](int x) { return x > a+c;};
10      [a,c = b+1](int x) { return x > a+c;};
11  }
```

Este código devuelve un total de cinco mutantes para este operador. En la siguiente lista se pueden observar dichos mutantes.

```
The mutant 'm51_1_1_Pruebas_LMB' has been created.
The mutant 'm51_2_1_Pruebas_LMB' has been created.
The mutant 'm51_3_1_Pruebas_LMB' has been created.
The mutant 'm51_4_1_Pruebas_LMB' has been created.
The mutant 'm51_5_1_Pruebas_LMB' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente.

1. El mutante m51_1_1_Pruebas_LMB contiene la siguiente mutación en la línea 6: **[&](int x) { return x > a+b;};**.
2. El mutante m51_2_1_Pruebas_LMB contiene la siguiente mutación en la línea 7: **[&a, &b](int x) { return x > a+b;};**.
3. El mutante m51_3_1_Pruebas_LMB contiene la siguiente mutación en la línea 8: **[&, &b](int x) { return x > a+b;};**.
4. El mutante m51_4_1_Pruebas_LMB contiene la siguiente mutación en la línea 9: **[&a, c = b](int x) { return x > a+c;};**.
5. El mutante m51_5_1_Pruebas_LMB contiene la siguiente mutación en la línea 10: **[&a, c = b+1](int x) { return x > a+c;};**.

6.3. Pruebas sobre el operador FWD

La entrada para este operador se compone de una función main que llama a un forward, el cual se deberá modificar por el operador FWD.

```
1   #include <utility>
2   #include <iostream>
3
4   void overloaded (const int& x) {std::cout << "[lvalue]";}
5   void overloaded (int&& x) {std::cout << "[rvalue]";}
6
7   template <class T> void fn (T&& x) {
8     overloaded (x);
9     overloaded (std::forward<T>(x));
10  }
```

```
11
12  int main () {
13    int a;
14    fn (0);
15    std::cout << '\n';
16
17    return 0;
18  }
```

Este código devuelve un total de un mutante para el operador que estamos probando. El mutante se puede observar en la siguiente lista.

```
The mutant 'm49_1_1_Pruebas_FWD' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente.

1. El mutante m49_1_1_Pruebas_FWD contiene la siguiente mutación en la línea 9: ***overloaded (std::move¡T¿(x));***.

6.4. Pruebas sobre el operador INI

La entrada para este operador se compone de una función main con una lista de llamadas a creación de vectores y un objeto con un contructor con lista de inicialización. Con este código podremos comprobar todas las funcionalidades del operador INI.

```
1   #include <iostream>
2   #include <vector>
3
4   template <class T>
5   struct S {
6       std::vector<T> v;
7       S(std::initializer_list<T> l) : v(l) {
8            std::cout << "constructed␣with␣a␣" << l.size() << "-element
                 ␣list\n";
9       }
10      void append(std::initializer_list<T> l) {
11          v.insert(v.end(), l.begin(), l.end());
12      }
13      std::pair<const T*, std::size_t> c_arr() const {
14          return {&v[0], v.size()};
15      }
16  };
17
18
19  int main() {
20      std::vector<int> v{0, 1, 2, 3, 4, 5};
21      std::vector<double> v2(2.0, 3.0);
22      S<int> s{1, 2, 3, 4, 5};
23  };
```

Este código devuelve tres mutantes que tienen asignados el identificador 50, ya que es el identificador asignado para el operador INI, el cual estamos probando.

```
The mutant 'm50_1_1_Pruebas_INI' has been created.
The mutant 'm50_2_1_Pruebas_INI' has been created.
The mutant 'm50_3_1_Pruebas_INI' has been created.
```

A continuación se puede observar la modificación que realiza cada mutante creado.

1. El mutante m50_1_1_Pruebas_INI contiene la siguiente mutación en la línea 20: ***std::vector<int> v(0, 1, 2, 3, 4, 5);***.
2. El mutante m50_2_1_Pruebas_INI contiene la siguiente mutación en la línea 21: ***std::vector<double> v2{2.0, 3.0};***.
3. El mutante m50_3_1_Pruebas_INI contiene la siguiente mutación en la línea 22: ***S<int> s(1, 2, 3, 4, 5);***.

Capítulo 7

Pruebas sobre nuevos operadores en programas reales

En este capítulo podemos encontrar las pruebas realizadas a todos y cada uno de los operadores de nueva inclusión en la herramienta MuCPP. Estas pruebas se han realizado sobre programas reales, los cuales, han sido utilizados en artículos como [15]. Estas pruebas están divididas por operador de mutación y se componen en primer lugar del código de entrada a MuCPP y la salida (mutantes) que produce. Posteriormente se incluye una lista con los mutantes erróneos generados y los no erróneos, estableciendo por último una tabla con los datos de forma más generalizada. Los operadores sobre los que se han realizado las pruebas pueden observarse en la tabla 66

Operador	**Descripción**
FOR	Operador de eliminación de referencia de rango en bucles for
LMB	Operador de paso por referencia en funciones lambda
FWD	Operador de modificación de llamadas a forward por llamadas a move
INI	Operador de llamada al constructor de inicialización con lista

Tabla 66: Tabla de operadores incluidos en MuCPP en el presente proyecto.

Por otro lado, los programas sobre los que se ha realizado las pruebas pueden ser observados en la tabla 67. Como se puede ver en dicha tabla, son programas de gran envergadura, con gran cantidad de clases y pruebas.

Proyecto	**Commit**	**Líneas de código**	**Líneas de tests**	**Número de commits**	**Grupo**
Corrade	ff3b351	6500	9100	1898	10
EntityX	6389b1f	9000	1000	296	28
Json	a09193e	8000	18000	1973	59
Antonie	59deb0d	9000	100	306	2

Tabla 67: Programas reales sobre los que se han realizado las pruebas de los nuevos operadores.

Los programas expresados en la tabla 67 serán descritos a continuación, incluyendo las acciones realizadas sobre los proyectos en cada caso para compilar y pasar las pruebas, comprobando si las pasa o no cada uno de los mutantes.

- **Proyecto Corrade.** Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas. Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

  Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza ctest para compilar y ejecutar sus pruebas.

- **Proyecto EntityX.** EntityX es un sistema de componentes de entidades que utiliza las características de C++11 para proporcionar una gestión de componentes de tipo seguro, entrega de eventos, etc. Se construyó durante la creación de un tirador espacial en 2D.

  Este proyecto ha sido implementado por un grupo de desarrolladores de 28 personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 6389b1f, también utilizado en [15], con un total de doscientos noventa y cinco commits anteriores a este.

  Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza un marco de pruebas propio, para más información puede visitar la web https://github.com/abeimler/ecs_benchmark.

- **Proyecto Json.** Json es una biblioteca de una sola cabecera para trabajar con Json en C++.

  Este proyecto ha sido implementado por un grupo de desarrolladores de 59 personas. Contempla un total de más de ocho mil líneas de código fuente desarrolladas y más de dieciocho mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit a09193e, también utilizado en [15], con un total de mil novecientos setenta y dos commits anteriores a este.

  Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza ctest para compilar y ejecutar sus pruebas.

- **Proyecto Antonie.** Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza boost para compilar y ejecutar sus pruebas. Para más información puede acceder a la web https://travis-ci.org/github/beaumontlab/antonie.

Destacar que los mutantes expresados en el artículo [15] no expresan los mutantes vivos, equivalentes o muertos de la misma forma que en el presente proyecto, dado que el propio autor indicó que estos mutantes no fueron ejecutados y las modificaciones se realizaron sin ningún tipo de herramienta automatizada. Es por ello que se realiza la siguiente aclaración:

- En el artículo [15], un mutante muerto es aquel que podría llegar a ser detectado por una prueba y uno vivo aquel que no hay forma de detectarlo (equivalente).
- En el presente proyecto, un mutante muerto es aquel detectado por las pruebas y uno vivo aquel que no es detectado, pero sin saber si esos mutantes son realmente equivalentes o simplemente faltan pruebas para detectarlos.

7.1. Pruebas sobre operador FOR

Una vez implementado el operador FOR (capítulo 5), y realizadas unas pruebas generales (capítulo 6), se considera necesaria su utilización en programas reales, con el fin de comprobar su funcionamiento real definitivamente. Para ello se ha hecho uso de los programas expresados en la tabla 67.

En las secciones sobre cada programa se dispondrán dos gráficas, en primer lugar, una gráfica sobre información de los mutantes generados sobre el programa real concreto por el operador implementado. En segundo lugar, se dispondrá una gráfica comparativa entre el operador implementado y el operador que se puede encontrar en el artículo [15].

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FOR implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

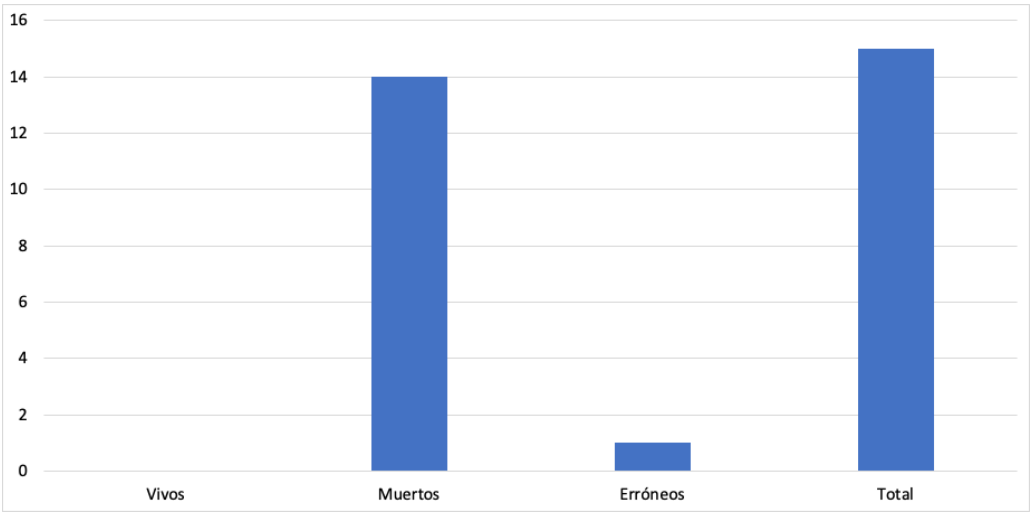

Figura 4: Gráfica salida operador FOR sobre proyecto Corrade.

Como se puede observar en la figura 4, se han obtenido un total de quince mutantes en el proyecto Corrade, de los cuales uno de ellos resultó erróneo y los catorce restantes muertos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

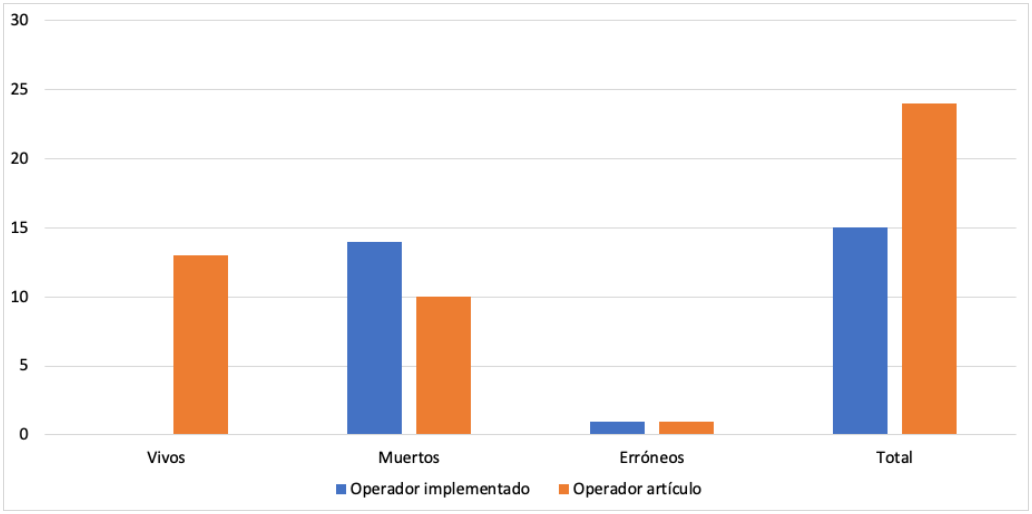

Figura 5: Gráfica comparativa resultados operador FOR sobre proyecto Corrade implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 5 se han reducido el número de mutantes equivalentes, no aportando ningún mutante vivo o equivalente, llevándolos a cero. Se ha incrementado en

uno el número de mutantes válidos, aunque dicho mutante también resultó muerto.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido reducir el tiempo de espera en mutantes poco significativos y sin valor, como son los mutantes equivalentes.

Proyecto EntityX

EntityX es un sistema de componentes de entidades que utiliza las características de C++11 para proporcionar una gestión de componentes de tipo seguro, entrega de eventos, etc. Se construyó durante la creación de un tirador espacial en 2D.

Este proyecto ha sido implementado por un grupo de desarrolladores de 28 personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 6389b1f, también utilizado en [15], con un total de doscientos noventa y cinco commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FOR implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

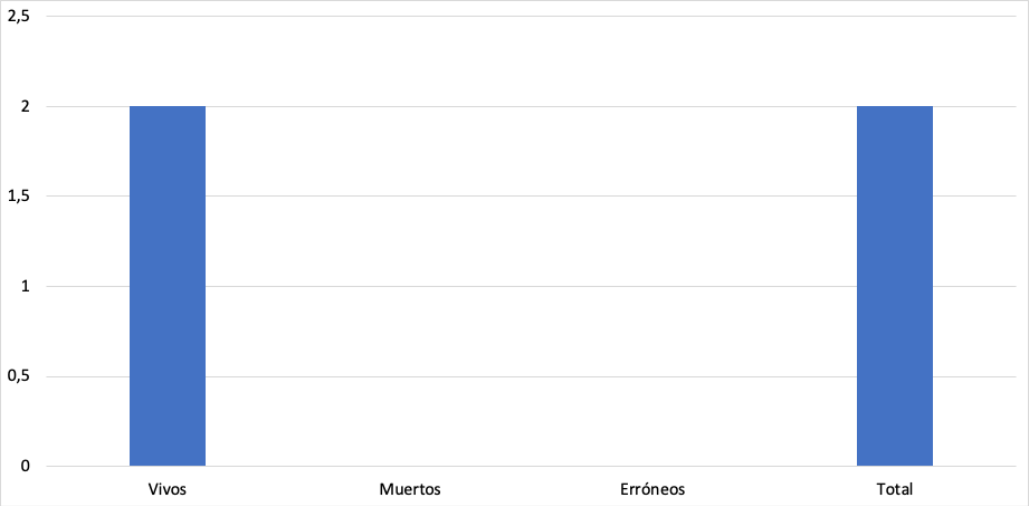

Figura 6: Gráfica salida operador FOR sobre proyecto EntityX.

Como se puede observar en la figura 6, se han obtenido un total de dos mutantes en el proyecto EntityX, los cuales han resultado vivos tras pasarle las pruebas correspondientes.

Cabe destacar que este proyecto contempla una cantidad de pruebas insuficientes, con una calidad baja, como para no poder matar a todos los mutantes propuestos por el operador.

Figura 7: Gráfica comparativa resultados operador FOR sobre proyecto EntityX implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 7 se ha mantenido el número de mutantes vivos.

Cabe destacar que aunque se ha realizado una mejora que se ha podido observar en otros proyectos, esta mejora no afecta a la calidad inicial de creación de mutantes.

Proyecto Json

Json es una biblioteca de una sola cabecera para trabajar con Json en C++.

Este proyecto ha sido implementado por un grupo de desarrolladores de 59 personas. Contempla un total de más de ocho mil líneas de código fuente desarrolladas y más de dieciocho mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit a09193e, también utilizado en [15], con un total de mil novecientos setenta y dos commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FOR implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

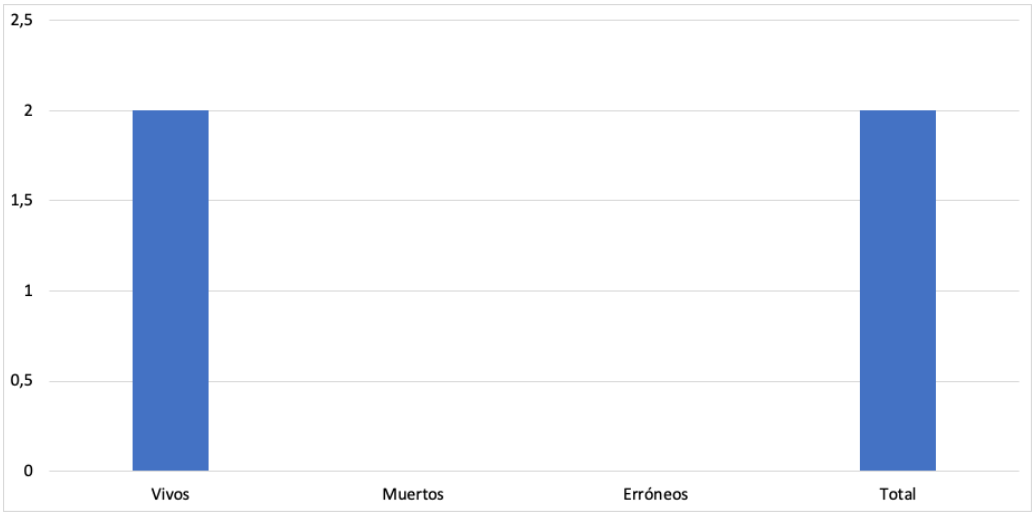

Figura 8: Gráfica salida operador FOR sobre proyecto Json.

Como se puede observar en la figura 8, se han obtenido un total de dos mutantes en el proyecto Json, los cuales han resultado vivos tras su paso por las pruebas creadas por el grupo de desarrollo.

Cabe destacar que este proyecto contempla una cantidad de pruebas insuficiente, o dichas pruebas son de poca calidad, como para no poder matar a los mutantes propuestos por el operador.

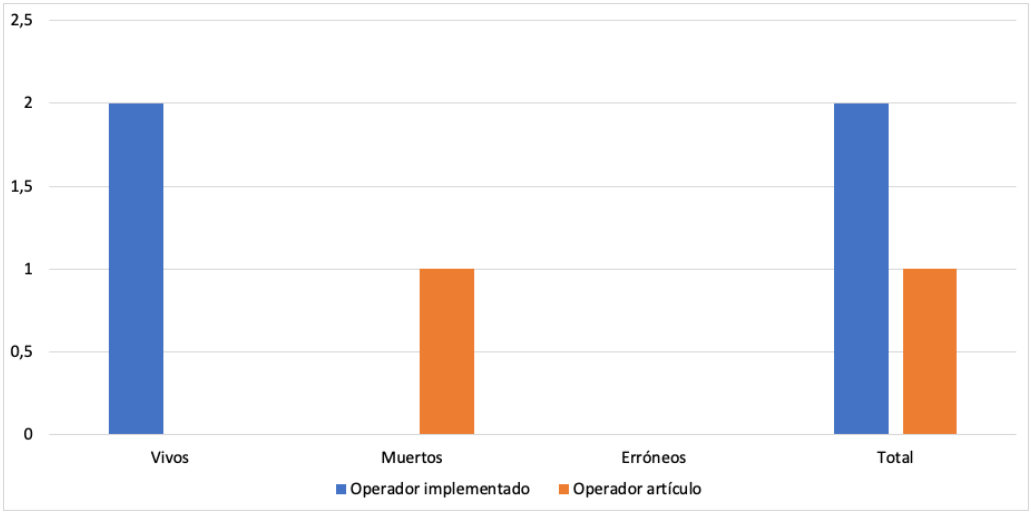

Figura 9: Gráfica comparativa resultados operador FOR sobre proyecto Json implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 9 se ha aumentado por un lado el número de mutantes, pasando de uno a dos. Por otro lado, destacar que ambos mutantes encontrados en la

prueba realizada quedaron vivos, comparado con los resultados obtenidos en el artículo que los calificaba como muertos.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido reducir el número de mutantes que localizan las pruebas y aumentar el número generado, por lo que se contempla que el operador es de mejor calidad que su predecesor.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FOR implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

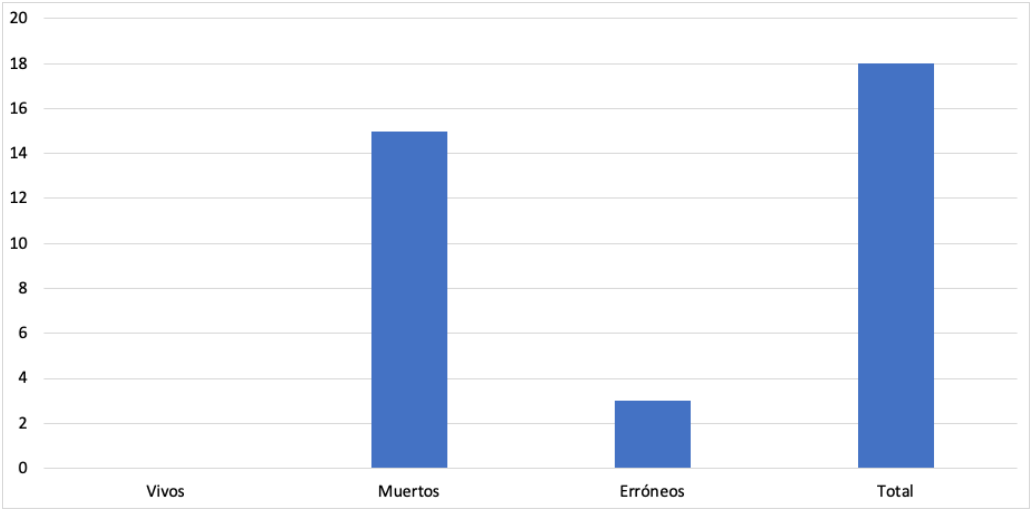

Figura 10: Gráfica salida operador FOR sobre proyecto Antonie.

Como se puede observar en la figura 10, se han obtenido un total de dieciocho mutantes en el proyecto Antonie, de los cuales quince de ellos resultaron muertos y tres erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

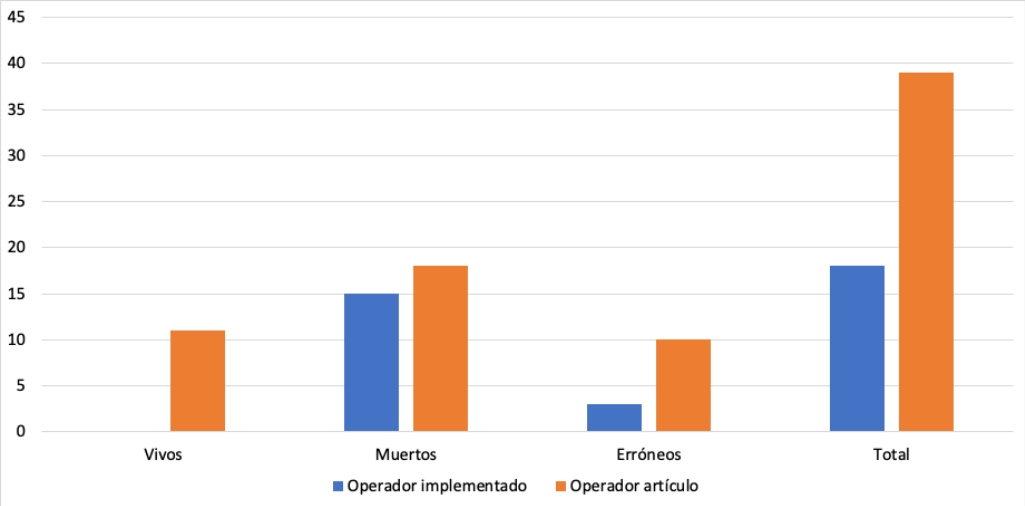

Figura 11: Gráfica comparativa resultados operador FOR sobre proyecto Antonie implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 11 se han reducido el número de mutantes equivalentes, no aportando ningún mutante vivo o equivalente, llevándolos a cero. Se ha disminuido el número de mutantes muertos, pasando de dieciocho a quince, y el número de mutantes erróneos, pasando de diez a tres.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido reducir el tiempo de espera en mutantes poco significativos y sin valor, como son los mutantes equivalentes.

Discusión sobre el operador FOR

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	15	1	0	14	100 %
EntityX	2	0	2	0	N/A
Json	2	0	2	0	0 %
Antonie	18	3	0	15	100 %

Tabla 68: Tabla resumen de resultados obtenidos con el operador FOR.

Para la realización de este análisis se ha hecho uso de los resultados obtenidos en el artículo [15] y se va a proceder a realizar una comparativa, determinando si el operador ha sido mejor implementado que en el propio artículo o no y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, mejora significativamente al operador implementado inicialmente por Parsai et al., disminuyendo el número de mutantes equivalentes, aumentando el número de mutantes muertos y disminuyendo el número de mutantes erróneos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de

un operador de estas caracteristicas.

7.2. Pruebas sobre operador LMB

Una vez implementado el operador LMB (capítulo 5), y realizadas unas pruebas generales (capítulo 6), se considera necesaria su utilización en programas reales, con el fin de comprobar su funcionamiento real definitivamente. Para ello se ha hecho uso de los programas expresados en la tabla 67.

En las secciones sobre cada programa se dispondrán dos gráficas, en primer lugar, una gráfica sobre información de los mutantes generados sobre el programa real concreto por el operador implementado. En segundo lugar, se dispondrá una gráfica comparativa entre el operador implementado y el operador que se puede encontrar en el artículo [15].

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Proyecto EntityX

EntityX es un sistema de componentes de entidades que utiliza las características de C++11 para proporcionar una gestión de componentes de tipo seguro, entrega de eventos, etc. Se construyó durante la creación de un tirador espacial en 2D.

Este proyecto ha sido implementado por un grupo de desarrolladores de 28 personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 6389b1f, también utilizado en [15], con un total de doscientos noventa y cinco commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador LMB implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

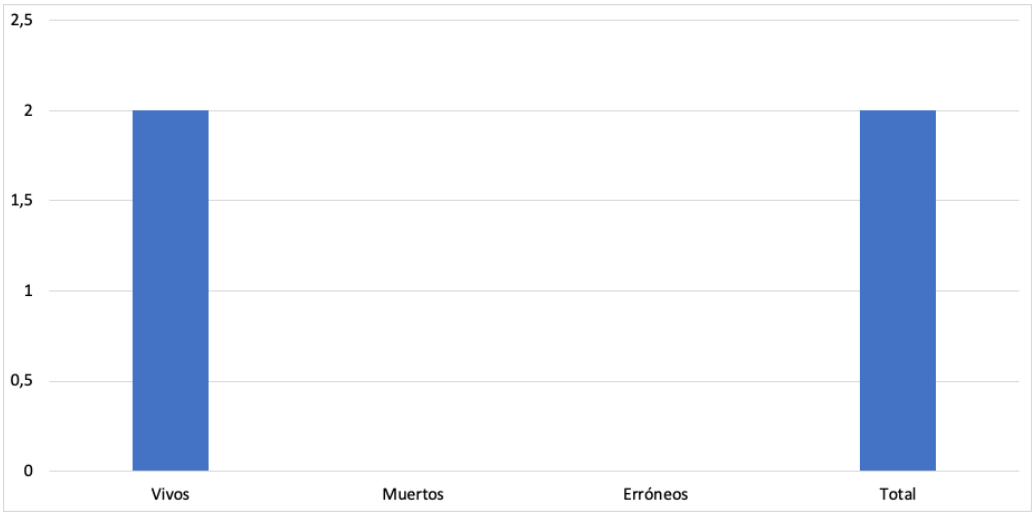

Figura 12: Gráfica salida operador LMB sobre proyecto EntityX.

Como se puede observar en la figura 12, se han obtenido un total de dos mutantes en el proyecto EntityX, los cuales han resultado vivos.

Cabe destacar que este proyecto contempla una cantidad de pruebas isuficientes, con una mala calidad, como para no poder matar a todos los mutantes propuestos por el operador.

Figura 13: Gráfica comparativa resultados operador LMB sobre proyecto EntityX implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 13 se ha aumentado el número de mutantes creados, pasando de cero a dos.

Cabe destacar que gracias a la mejora implementada en el operador en comparación

con su predecesor, se ha podido obtener mutantes en el paso del operador LMB sobre el proyecto.

Proyecto Json

Json es una biblioteca de una sola cabecera para trabajar con Json en C++.

Este proyecto ha sido implementado por un grupo de desarrolladores de 59 personas. Contempla un total de más de ocho mil líneas de código fuente desarrolladas y más de dieciocho mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit a09193e, también utilizado en [15], con un total de mil novecientos setenta y dos commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador LMB implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

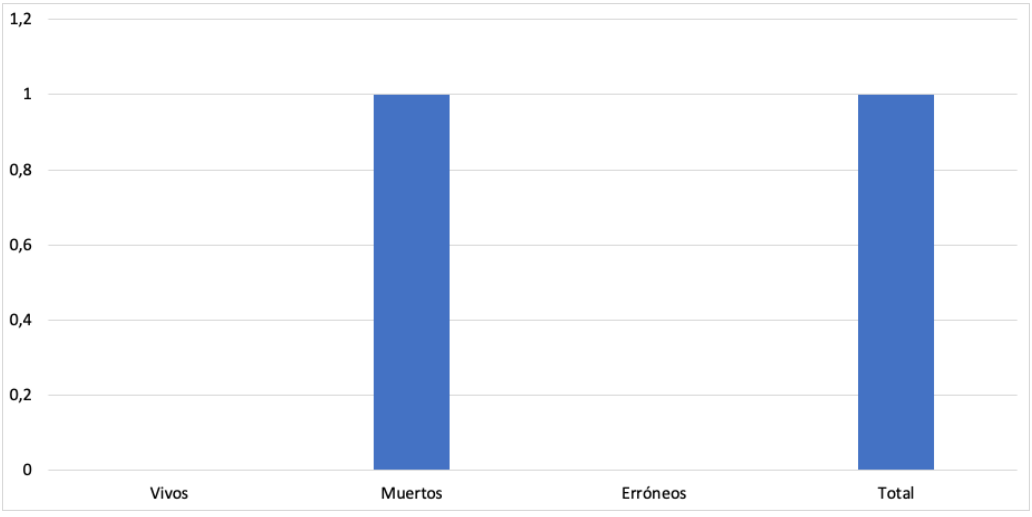

Figura 14: Gráfica salida operador LMB sobre proyecto Antonie.

Como se puede observar en la figura 14, se han obtenido un total de diecisiete mutante en el proyecto Antonie, los cuales han resultado erróneos.

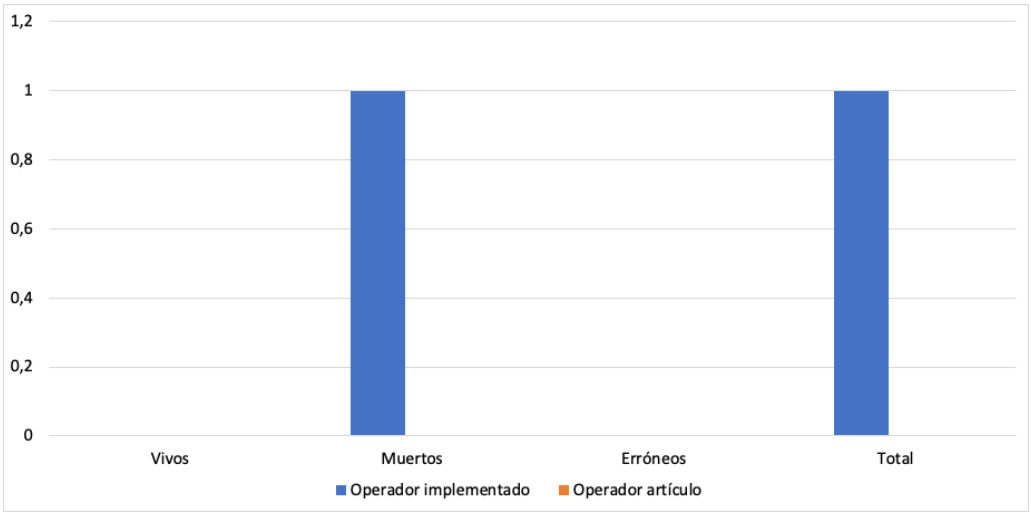

Figura 15: Gráfica comparativa resultados operador LMB sobre proyecto Antonie implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 15 el número de mutantes creados ha aumentado, aunque todos han resultado erróneos. En comparación con [15] en este se obtuvieron ocho mutantes vivos y seis mutantes muertos.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido obtener un mayor número de mutantes en el paso del operador LMB sobre el proyecto.

Discusión

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	0	0	0	0	N/A
EntityX	2	0	2	0	0 %
Json	0	0	0	0	N/A
Antonie	1	0	0	1	100 %

Para la realización de este análisis se ha hecho uso de los resultados obtenidos en el artículo [15] y se va a proceder a realizar una comparativa, determinando si el operador ha sido mejor implementado que en el propio artículo o no y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, mejora significativamente al operador implementado inicialmente por Parsai et al., pasando en todo momento de ningún operador a uno o dos operadores.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de un operador de estas características.

7.3. Pruebas sobre operador FWD

Una vez implementado el operador FWD (capítulo 5), y realizadas unas pruebas generales (capítulo 6), se considera necesaria su utilización en programas reales, con el fin de comprobar su funcionamiento real definitivamente. Para ello se ha hecho uso de los programas expresados en la tabla 67.

En las secciones sobre cada programa se dispondrán dos gráficas, en primer lugar, una gráfica sobre información de los mutantes generados sobre el programa real concreto por el operador implementado. En segundo lugar, se dispondrá una gráfica comparativa entre el operador implementado y el operador que se puede encontrar en el artículo [15].

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Cabe destacar que aunque este operador no haya creado ningún mutante, en el artículo de Parsai et al. se localizan un total de cinco mutantes, los cuales son clasificados como muertos. A continuación se muestra la gráfica comparativa.

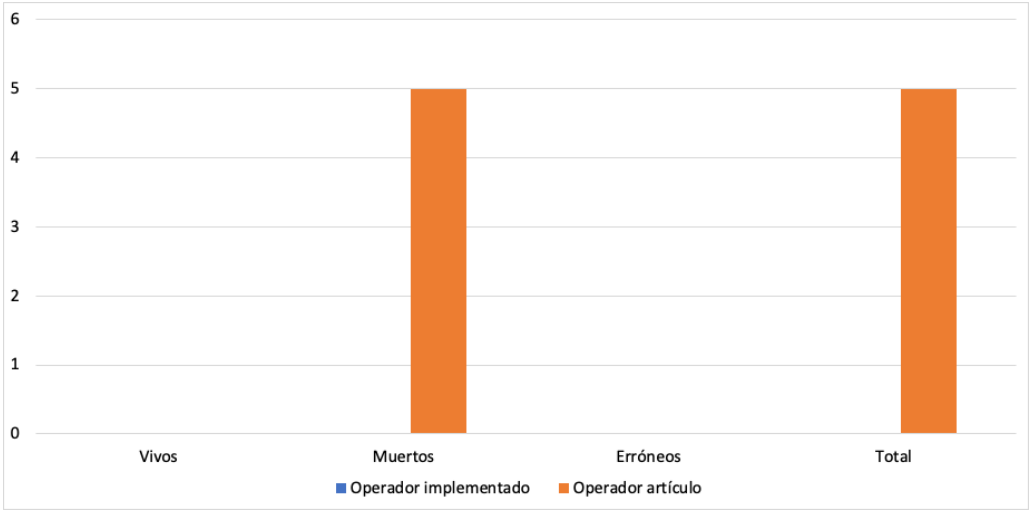

Figura 16: Gráfica comparativa resultados operador FWD sobre proyecto Corrade implementado y en artículo.

Proyecto EntityX

EntityX es un sistema de componentes de entidades que utiliza las características de C++11 para proporcionar una gestión de componentes de tipo seguro, entrega de eventos, etc. Se construyó durante la creación de un tirador espacial en 2D.

Este proyecto ha sido implementado por un grupo de desarrolladores de 28 personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 6389b1f, también utilizado en [15], con un total de doscientos noventa y cinco commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FWD implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

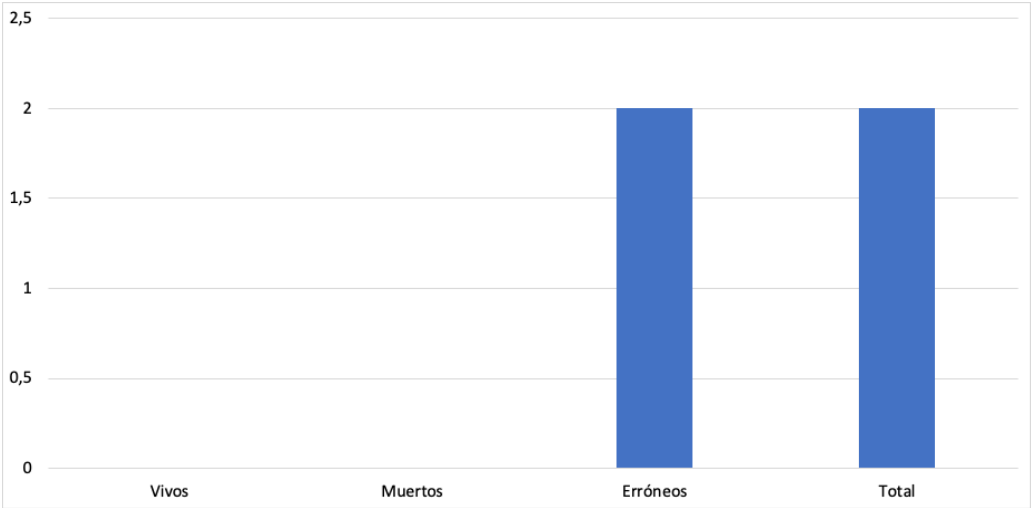

Figura 17: Gráfica salida operador FWD sobre proyecto EntityX.

Como se puede observar en la figura 17, se han obtenido un total de dos mutantes en el proyecto EntityX, los cuales han resultado erróneos.

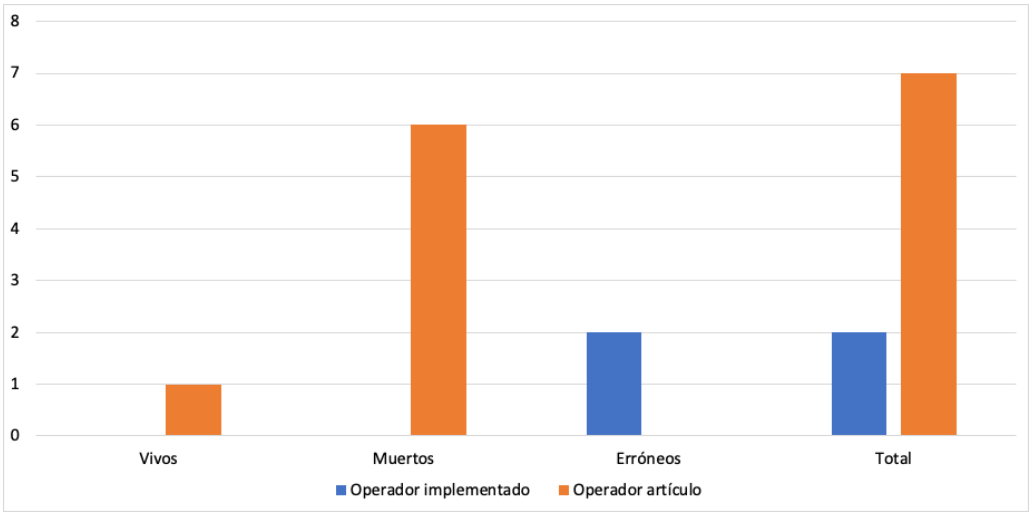

Figura 18: Gráfica comparativa resultados operador FWD sobre proyecto EntityX implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 18 se ha disminuido el número de mutantes creados y todos han resultado erróneos, en comparación con los seis muertos y uno vivo que se expresa en el artículo. Esta diferencia puede radicar en que en el artículo no se expresa en ningún momento que dichos mutantes hayan sido ejecutados.

Proyecto Json

Json es una biblioteca de una sola cabecera para trabajar con Json en C++.

Este proyecto ha sido implementado por un grupo de desarrolladores de 59 personas. Contempla un total de más de ocho mil líneas de código fuente desarrolladas y más de dieciocho mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit a09193e, también utilizado en [15], con un total de mil novecientos setenta y dos commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador FWD implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

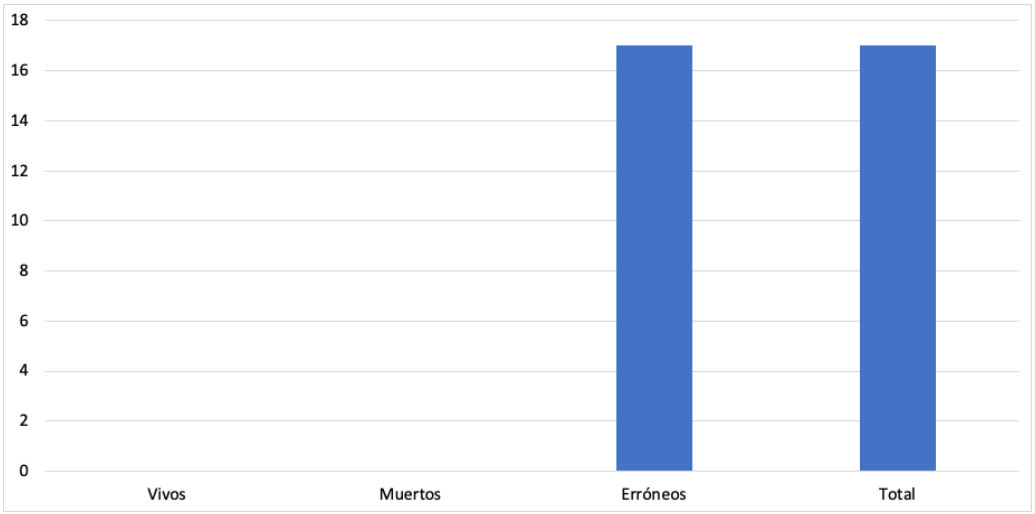

Figura 19: Gráfica salida operador FWD sobre proyecto Json.

Como se puede observar en la figura 19, se han obtenido un total de un mutante en el proyecto Json, el cual ha resultado muerto.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar al mutante propuesto por el operador.

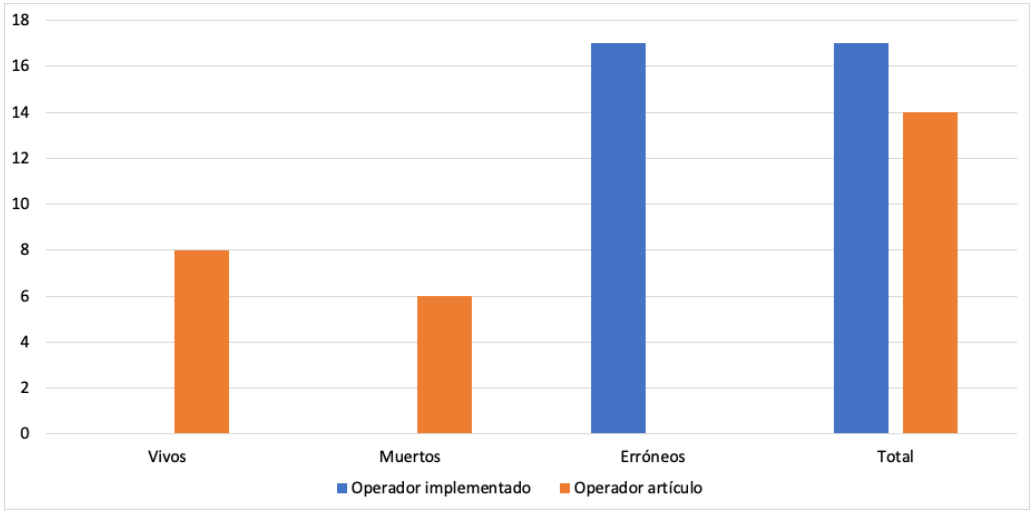

Figura 20: Gráfica comparativa resultados operador FWD sobre proyecto Json implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica **??** se han reducido el número de mutantes equivalentes, no aportando ningún mutante vivo o equivalente, llevándolos a cero. Se ha disminuido el número de mutantes muertos, pasando de dieciocho a quince, y el número de mutantes erróneos, pasando de diez a tres.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido reducir el tiempo de espera en mutantes poco significativos y sin valor, como son los mutantes equivalentes.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Discusión y comparativa

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	0	0	0	0	N/A
EntityX	2	2	0	0	N/A
Json	17	17	0	0	N/A
Antonie	0	0	0	0	N/A

Para la realización de este análisis se ha hecho uso de los resultados obtenidos en el artículo [15] y se va a proceder a realizar una comparativa, determinando si el operador ha sido mejor implementado que en el propio artículo o no y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, no mejora significativamente al operador implementado inicialmente por Parsai et al., pasando de operadores vivos o muertos a operadores erróneos. Cabe destacar que aunque en el artículo se expresan como mutantes equivalentes, inválidos y erróneos, no existe constancia de que dichos operadores fueran o no ejecutados.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta, tal y como se ha podido observar en el capítulo 6, con unas pruebas generales de uso.

7.4. Pruebas sobre operador INI

Una vez implementado el operador INI (capítulo 5), y realizadas unas pruebas generales (capítulo 6), se considera necesaria su utilización en programas reales, con el fin de comprobar su funcionamiento real definitivamente. Para ello se ha hecho uso de los programas expresados en la tabla 67.

En las secciones sobre cada programa se dispondrán dos gráficas, en primer lugar, una gráfica sobre información de los mutantes generados sobre el programa real concreto por el operador implementado. En segundo lugar, se dispondrá una gráfica comparativa entre el operador implementado y el operador que se puede encontrar en el artículo [15].

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataINIma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador INI implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

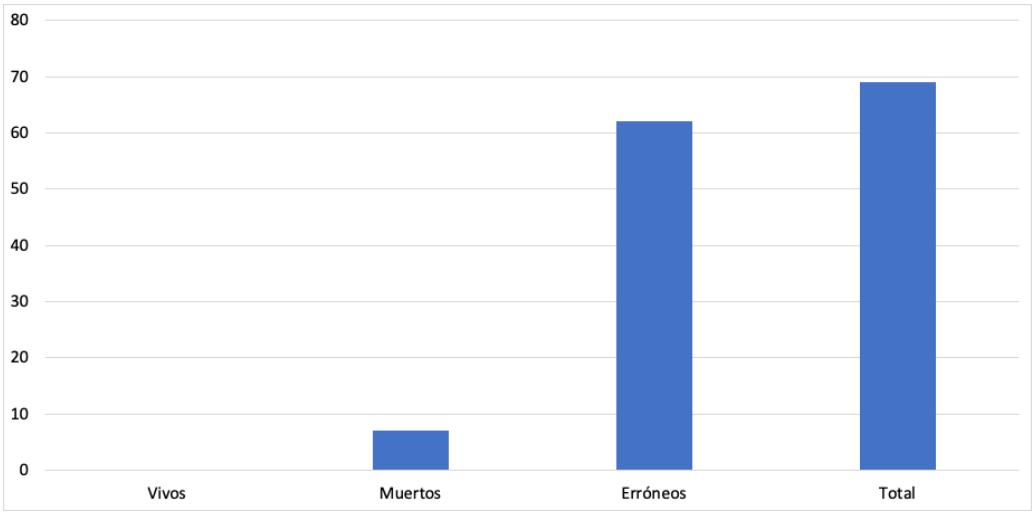

Figura 21: Gráfica salida operador INI sobre proyecto Corrade.

Como se puede observar en la figura 21, se han obtenido un total de sesenta y nueve mutantes en el proyecto Corrade, de los cuales sesenta y dos resultaron erróneos y los siete restantes muertos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

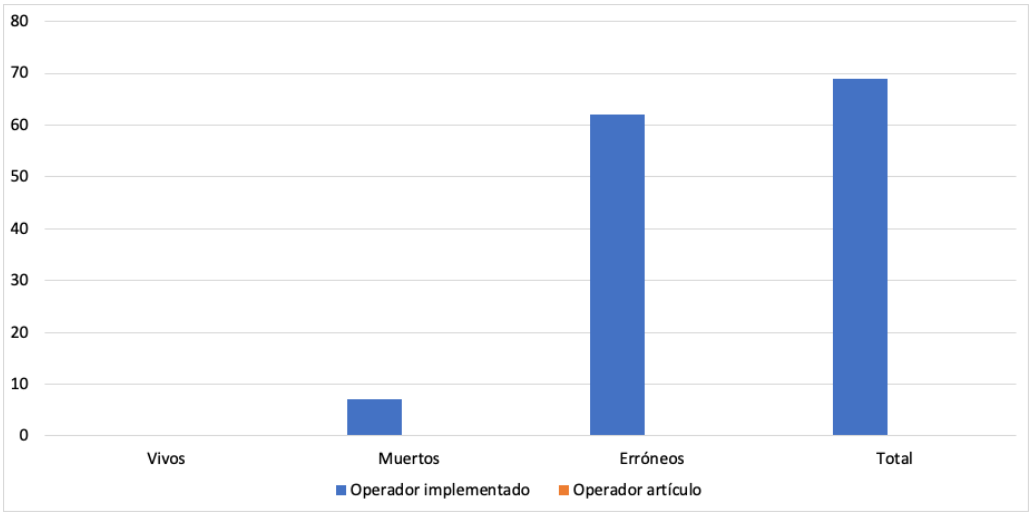

Figura 22: Gráfica comparativa resultados operador INI sobre proyecto Corrade implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 22, se ha pasado de no generar ningún mutante a generar sesenta y nueve mutantes.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido hacer uso del operador en el proyecto.

Proyecto EntityX

EntityX es un sistema de componentes de entidades que utiliza las características de C++11 para proporcionar una gestión de componentes de tipo seguro, entrega de eventos, etc. Se construyó durante la creación de un tirador espacial en 2D.

Este proyecto ha sido implementado por un grupo de desarrolladores de 28 personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 6389b1f, también utilizado en [15], con un total de doscientos noventa y cinco commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador INI implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

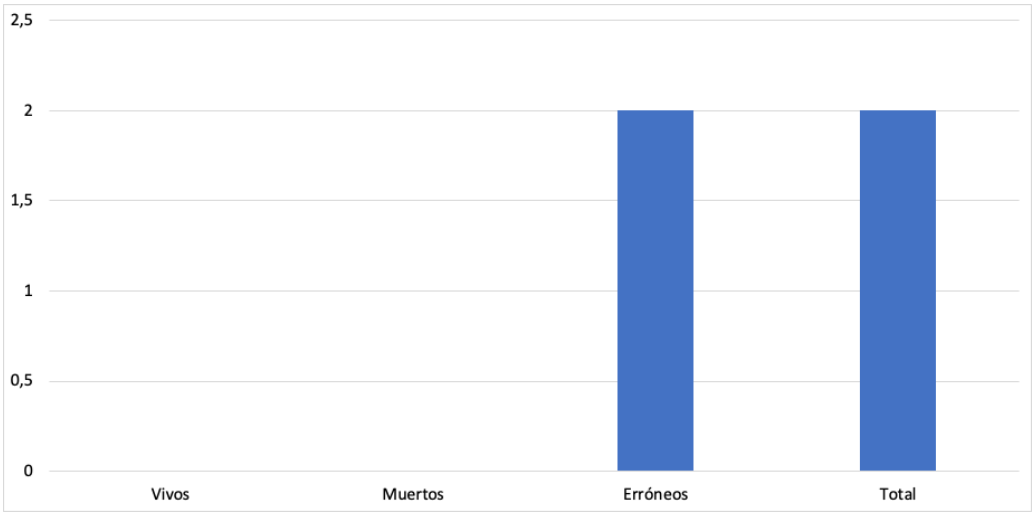

Figura 23: Gráfica salida operador INI sobre proyecto EntityX.

Como se puede observar en la figura 23, se han obtenido un total de dos mutantes en el proyecto EntityX, los cuales han resultado erróneos tras su intento de compilación.

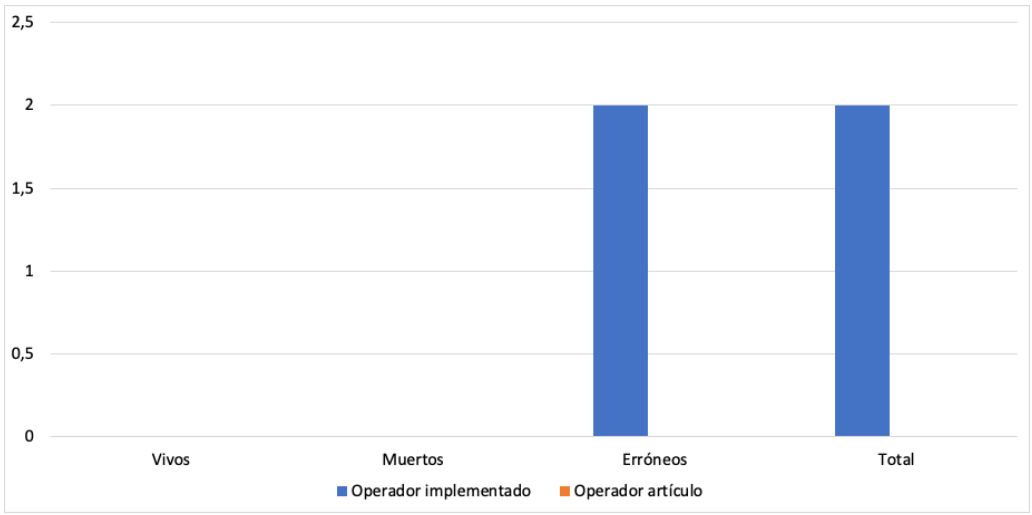

Figura 24: Gráfica comparativa resultados operador INI sobre proyecto EntityX implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 24 se ha aumentado el número de mutantes creados. Cabe destacar que los mutantes han creados han sido erróneos.

Proyecto Json

Json es una biblioteca de una sola cabecera para trabajar con Json en C++.

Este proyecto ha sido implementado por un grupo de desarrolladores de 59 personas. Contempla un total de más de ocho mil líneas de código fuente desarrolladas y más de dieciocho mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit a09193e, también utilizado en [15], con un total de mil novecientos setenta y dos commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador INI implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

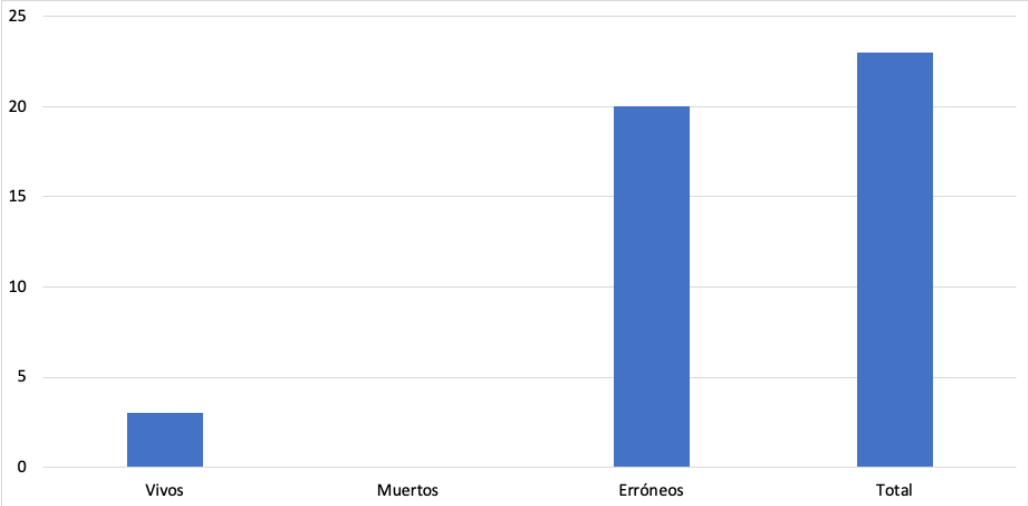

Figura 25: Gráfica salida operador INI sobre proyecto Json.

Como se puede observar en la figura 25, se han obtenido un total de veintitrés mutantes en el proyecto Json, de los cuales veinte de ellos han resultado erróneos y tres vivos.

Cabe destacar que este proyecto contempla una cantidad de pruebas insuficiente, o dichas pruebas son de poca calidad, como para no poder matar a los mutantes propuestos por el operador.

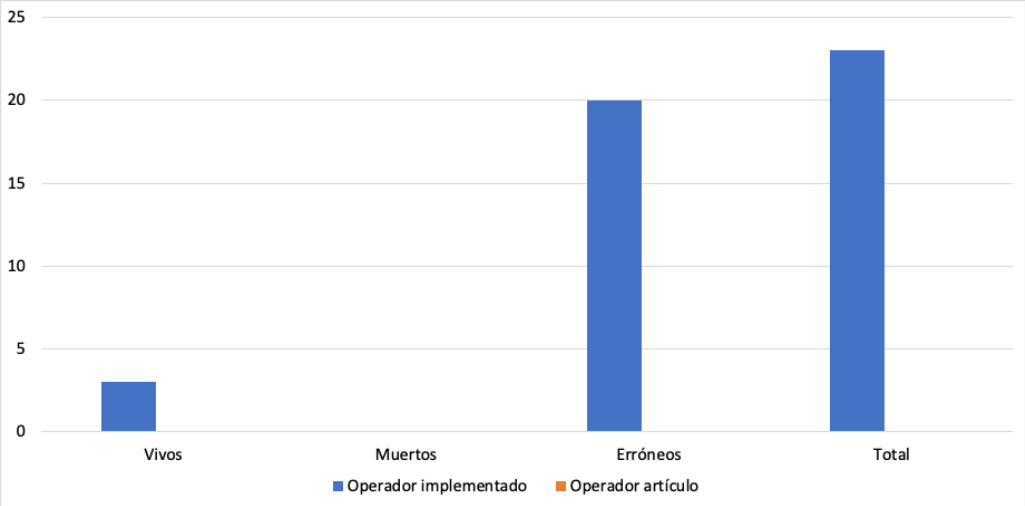

Figura 26: Gráfica comparativa resultados operador INI sobre proyecto Json implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 26 se ha aumentado el número de mutantes de forma significativa, pasando de ningún mutante a veintitrés.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se han podido obtener mutantes que no son localizados por las pruebas.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrán observar dos gráficas, una gráfica que contemplará los resultados obtenidos sobre este programa con el operador INI implementado en MuCPP y otra gráfica comparativa sobre los resultados obtenidos en el presente proyecto y los obtenidos por Parsai et al. en [15].

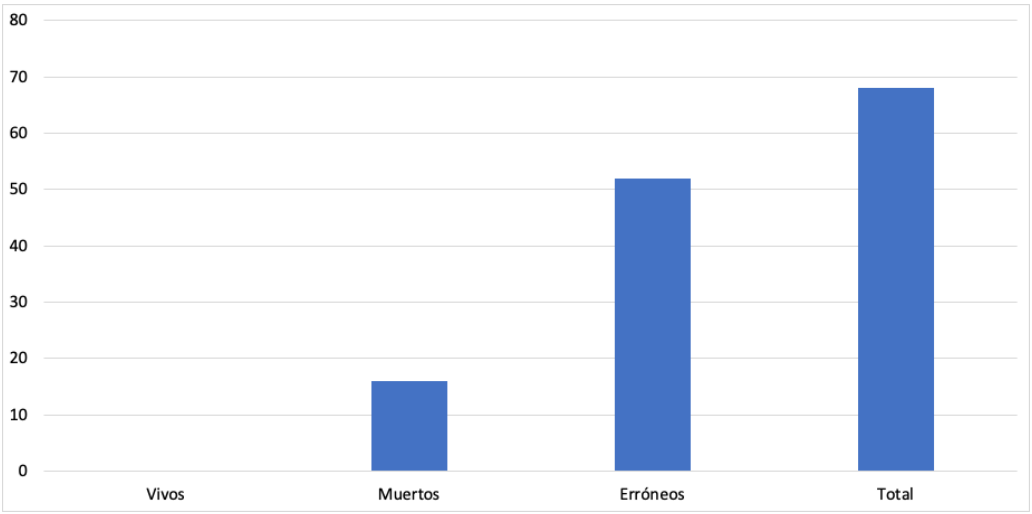

Figura 27: Gráfica salida operador INI sobre proyecto Antonie.

Como se puede observar en la figura 27, se han obtenido un total de sesenta y dos mutantes en el proyecto Antonie, de los cuales dieciséis de ellos resultaron muertos y cincuenta y dos erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

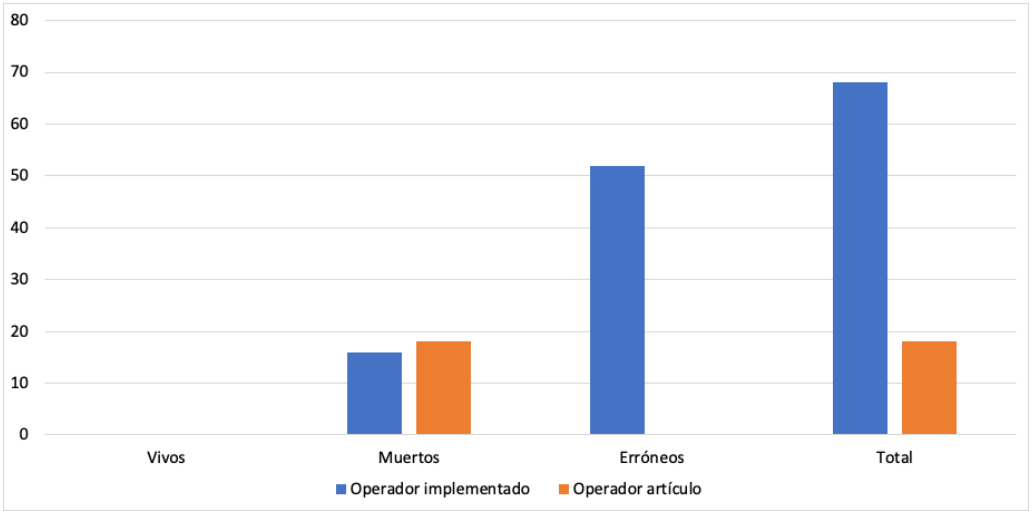

Figura 28: Gráfica comparativa resultados operador INI sobre proyecto Antonie implementado y en artículo.

En comparación con los resultados obtenidos en el artículo expuesto, tal y como se puede observar en la gráfica 28 se ha aumentado el número de mutantes de forma significativa, pasando de dieciocho a sesenta y dos.

Cabe destacar que gracias a la mejora implementada en el operador en comparación con su predecesor, se ha podido aumentar el número de mutantes obtenidos para esta operador y proyecto.

Discusión

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	69	62	0	7	100 %
EntityX	2	2	0	0	N/A
Json	23	20	3	0	0 %
Antonie	68	52	0	16	100 %

Para la realización de este análisis se ha hecho uso de los resultados obtenidos en el artículo [15] y se va a proceder a realizar una comparativa, determinando si el operador ha sido mejor implementado que en el propio artículo o no y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, mejora significativamente al operador implementado inicialmente por Parsai et al., pasando de ningún mutante en muchos casos a un gran número de mutantes.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta.

Capítulo 8

Pruebas sobre operadores incluidos en los últimos años en programas reales

En este capítulo podemos encontrar las pruebas realizadas a todos y cada uno de los operadores incluidos en los últimos años en la herramienta MuCPP. Estas pruebas se han realizado sobre programas reales, los cuales, han sido utilizados en artículos como [15]. Estas pruebas están divididas por operador de mutación y se componen en primer lugar del código de entrada a MuCPP y la salida (mutantes) que produce. Posteriormente se incluye una lista con los operadores erróneos generados y los no erróneos, estableciendo por último una tabla con los datos de forma más generalizada. Los operadores que a los que se han realizado las pruebas pueden observarse en la tabla 69.

Operador	**Descripción**
SOR	Operador de reemplazo de operadores de desplazamiento
FOR	Operador de eliminación de referencia de rango en bucles for
LMB	Operador de paso por referencia en funciones lambda
FWD	Operador de modificación de llamadas a forward por llamadas a move
INI	Operador de llamada al constructor de inicialización con lista

Tabla 69: Tabla de operadores incluidos en MuCPP en los últimos años.

Por otro lado, los programas expresados en la tabla 70 serán descritos a continuación, incluyendo las acciones realizadas sobre los proyectos en cada caso para compilar y pasar las pruebas, comprobando si las pasa o no cada uno de los mutantes.

Proyecto	**Commit**	**Líneas de código**	**Líneas de tests**	**Número de commits**	**Grupo**
Corrade	ff3b351	6500	9100	1898	10
Antonie	59deb0d	9000	100	306	2

Tabla 70: Programas reales sobre los que se han realizado las pruebas de los operadores incluidos en los últimos años.

- **Proyecto Corrade.** Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas. Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

  Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza ctest para compilar y ejecutar sus pruebas.

- **Proyecto Antonie.** Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

  Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

  Destacar que para la realización de las pruebas de este operador ha sido necesario crear un programa en C++ de compilación, obtención de resultados de ejecución y pruebas y generación de resultados. Para ello ha sido necesario previamente analizar la forma de compilación y sus resultados, la ejecución y sus resultados, las compilación de las pruebas, la ejecución de estas y los resultados que aportan. Cabe destacar que este proyecto utiliza boost para compilar y ejecutar sus pruebas. Para más información puede acceder a la web https://travis-ci.org/github/beaumontlab/antonie.

Destacar que los mutantes expresados en el artículo [15] no expresan los mutantes vivos, equivalentes o muertos de la misma forma que en el presente proyecto, dado que el propio autor indicó que estos mutantes no fueron ejecutados y las modificaciones se realizaron sin ningún tipo de herramienta automatizada. Es por ello que se realiza la siguiente aclaración:

- En el artículo [15], un mutante muerto es aquel que podría llegar a ser detectado por una prueba y uno vivo aquel que no hay forma de detectarlo (equivalente).

- En el presente proyecto, un mutante muerto es aquel detectado por las pruebas y uno vivo aquel que no es detectado, pero sin saber si esos mutantes son realmente equivalentes o simplemente faltan pruebas para detectarlos.

8.1. Pruebas sobre operador SOR

Tal y como se ha explicado anteriormente se ha realizado las pruebas sobre diferentes programas que serán expresados en cada uno de ellos en las subsecciones siguientes, llegando a las conclusiones en su subsección concreta.

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador SOR implementado en MuCPP.

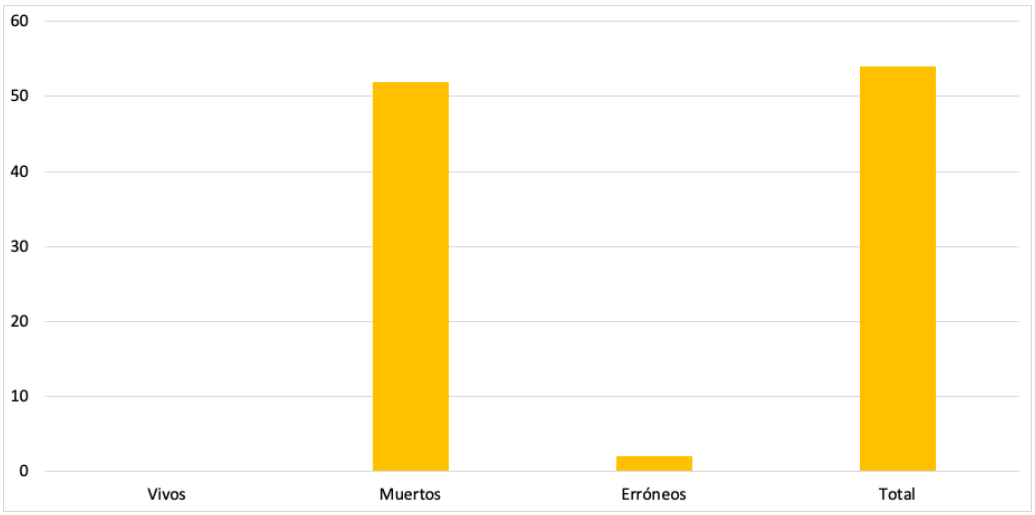

Figura 29: Gráfica salida operador SOR sobre proyecto Corrade.

Como se puede observar en la figura 29, se han obtenido un total de cinco mutantes en el proyecto Corrade, de los cuales cuatro de ellos resultaron erróneos y el restante muerto.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrá observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador SOR implementado en MuCPP.

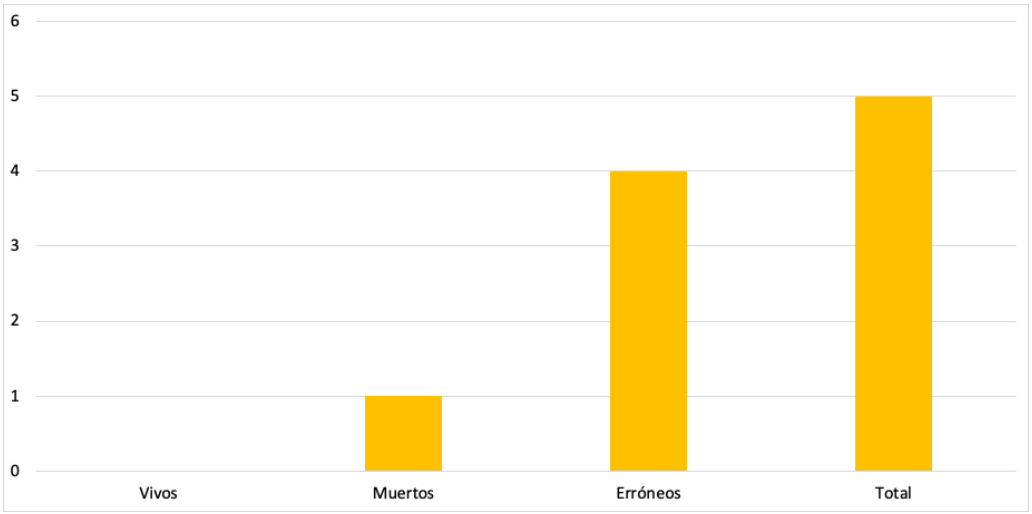

Figura 30: Gráfica salida operador SOR sobre proyecto Antonie.

Como se puede observar en la figura 30, se han obtenido un total de dieciocho mutantes en el proyecto Antonie, de los cuales quince de ellos resultaron muertos y tres erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

Discusión

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	54	2	0	52	100 %
Antonie	5	4	0	1	100 %

Tabla 71: Tabla operador SOR en MuCPP.

Para la realización de este análisis se ha hecho uso de los resultados obtenidos, determinando si el operador ha sido bien implementado y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, crea un número correcto de mutantes, no contemplando un número excesivo de operadores erróneos en comparación con el resto de tipos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de un operador de estas características.

8.2. Pruebas sobre operador SDL

Tal y como se ha explicado anteriormente se ha realizado las pruebas sobre diferentes programas que serán expresados en cada uno de ellos en las subsecciones siguientes, llegando a las conclusiones en su subsección concreta.

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador SDL implementado en MuCPP.

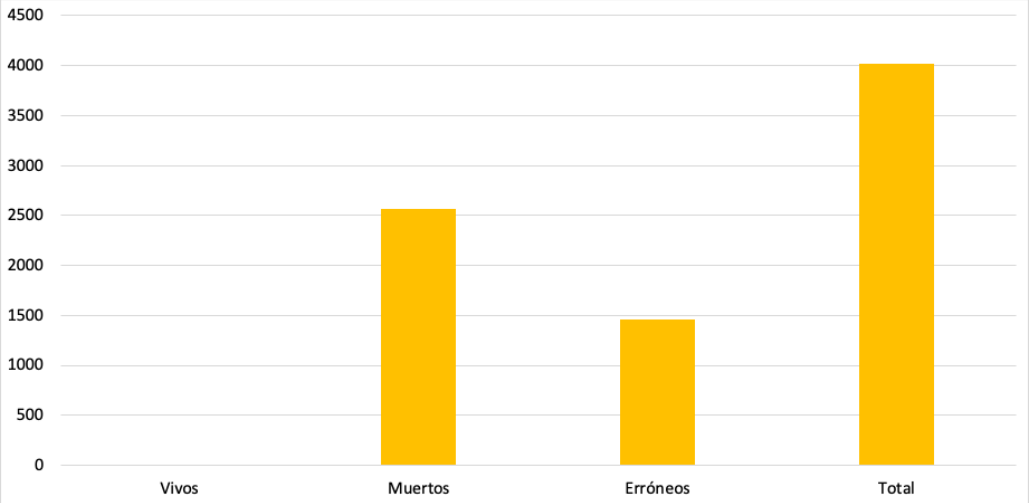

Figura 31: Gráfica salida operador SDL sobre proyecto Corrade.

Como se puede observar en la figura 32, se han obtenido un total de cuatro mil diecinueve mutantes en el proyecto Corrade, de los cuales dos mil quinientos sesenta y cuatro de ellos resultaron muertos y mil cuatrocientos cincuenta y cinco erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrá observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador SDL implementado en MuCPP.

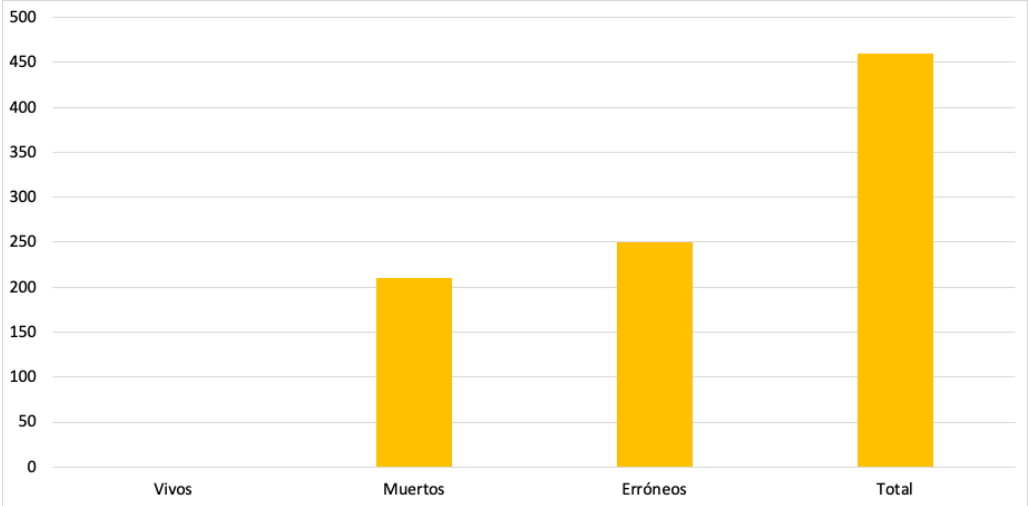

Figura 32: Gráfica salida operador SDL sobre proyecto Antonie.

Como se puede observar en la figura 32, se han obtenido un total de cuatrocientos sesenta mutantes en el proyecto Antonie, de los cuales doscientos diez de ellos resultaron muertos y doscientos cincuenta erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

Discusión

Tabla 72: Tabla operador SOR en MuCPP.

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	4019	1455	0	2564	100 %
Antonie	460	250	0	210	100 %

Para la realización de este análisis se ha hecho uso de los resultados obtenidos, determinando si el operador ha sido bien implementado y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, crea un número correcto de mutantes, no contemplando un número excesivo de operadores erróneos en comparación con el resto de tipos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de un operador de estas características.

8.3. Pruebas sobre operador ODL

Tal y como se ha explicado anteriormente se ha realizado las pruebas sobre diferentes programas que serán expresados en cada uno de ellos en las subsecciones siguientes, llegando a las conclusiones en su subsección concreta.

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

A continuación se podrán observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador ODL implementado en MuCPP.

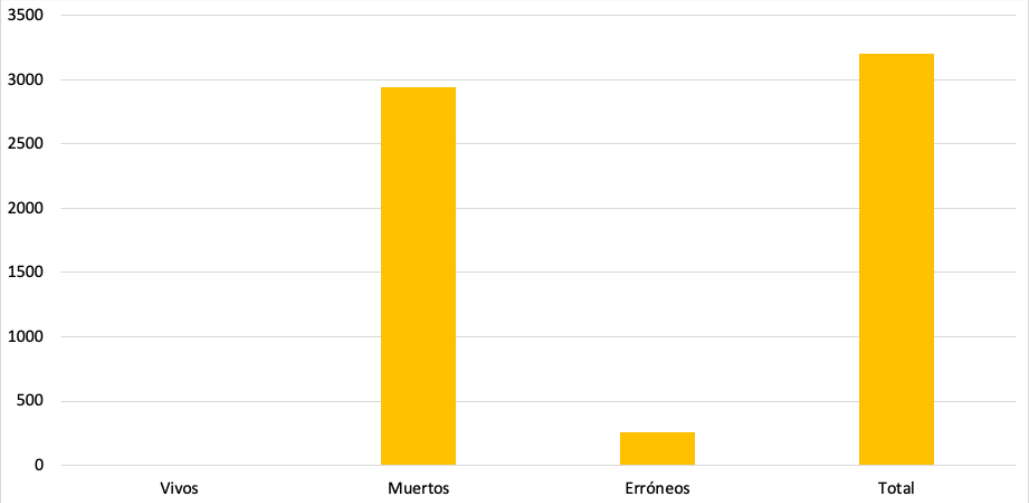

Figura 33: Gráfica salida operador ODL sobre proyecto Corrade.

Como se puede observar en la figura 34, se han obtenido un total de tres mil doscientos seis mutantes en el proyecto Corrade, de los cuales dos mil novecientos cuarenta y siete de ellos resultaron muertos y doscientos cincuenta y nueve erróneos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad correcta, como para poder matar a todos los mutantes propuestos por el operador.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrá observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador ODL implementado en MuCPP.

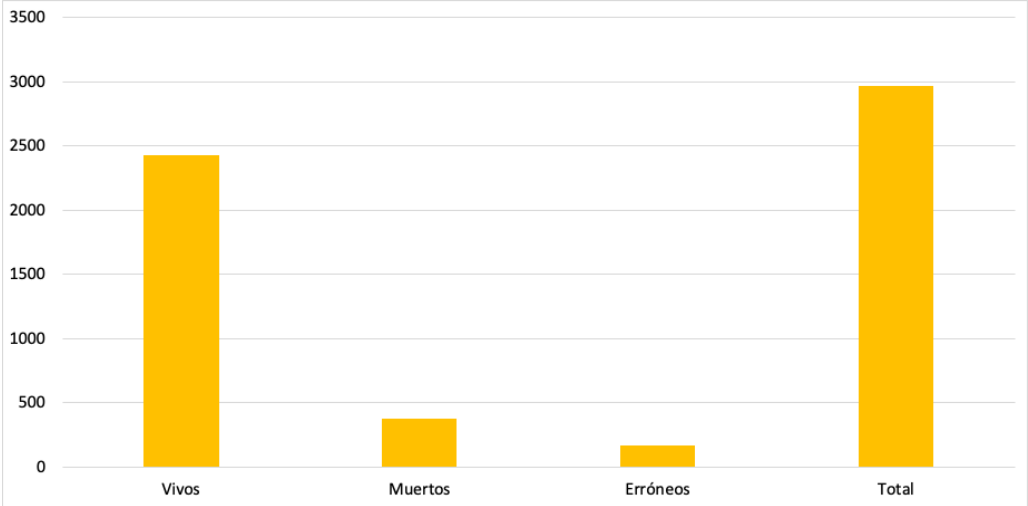

Figura 34: Gráfica salida operador ODL sobre proyecto Antonie.

Como se puede observar en la figura 34, se han obtenido un total de dos mil cuatrocientos treinta y dos mutantes en el proyecto Antonie, de los cuales trescientos setenta y tres de ellos resultaron muertos, ciento sesenta y cuatro erróneos y dos mil cuatrocientos treinta y dos vivos.

Cabe destacar que este proyecto contempla una cantidad de pruebas insuficientes, con una calidad baja, como para no poder matar a todos los mutantes propuestos por el operador.

Discusión

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	3206	259	0	2947	100 %
Antonie	2969	164	2432	373	13,3 %

Tabla 73: Tabla operador ODL en MuCPP.

Para la realización de este análisis se ha hecho uso de los resultados obtenidos, determinando si el operador ha sido bien implementado y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, crea un número correcto de mutantes, no contemplando un número excesivo de operadores erróneos en comparación con el resto de tipos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de un operador de estas características.

8.4. Pruebas sobre operador CSD

Tal y como se ha explicado anteriormente se ha realizado las pruebas sobre diferentes programas que serán expresados en cada uno de ellos en las subsecciones siguientes,

llegando a las conclusiones en su subsección concreta.

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Discusión

Tabla 74: Tabla operador SOR en MuCPP.

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	0	0	0	0	N/A
Antonie	0	0	0	0	N/A

Para la realización de este análisis se ha hecho uso de los resultados obtenidos, determinando si el operador ha sido bien implementado y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, no ha creado ningún mutante en los proyectos empleados, esto se debe a que la acción que realiza dicho operador es una acción muy concreta que claramente no se utiliza en todos los proyectos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta, tal y como se pudo observar previamente con un programa general creado para probar dicho operador en [28].

8.5. Pruebas sobre operador CSI

Tal y como se ha explicado anteriormente se ha realizado las pruebas sobre diferentes programas que serán expresados en cada uno de ellos en las subsecciones siguientes, llegando a las conclusiones en su subsección concreta.

Proyecto Corrade

Corrade es una biblioteca de utilidades multiplataforma escrita en C++11 y C++14. Se utiliza como base para el motor gráfico Magnum, entre otras cosas.

Este proyecto ha sido implementado por un grupo de desarrolladores de 10 personas. Contempla un total de más de seis mil quinientas líneas de código fuente desarrolladas y más de nueve mil líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit fff3b351, también utilizado en [15], con un total de mil ochocientos noventa y siete commits anteriores a este.

No se generan mutaciones para este operador y proyecto.

Proyecto Antonie

Antonie es un procesador integrado, robusto, fiable y rápido de lecturas de ADN, en su mayoría de plataformas de secuenciamiento de próxima generación. Actualmente se centra en los procarióticos y otros pequeños genomas.

Este proyecto ha sido implementado por un grupo de desarrolladores de dos personas. Contempla un total de más de nueve mil líneas de código fuente desarrolladas y más de cien líneas para la creación de tests. Para la realización de estas pruebas se ha hecho uso de su commit 59deb0d, también utilizado en [15], con un total de trescientos cinco commits anteriores a este.

A continuación se podrá observar una gráfica que contemplará los resultados obtenidos sobre este programa con el operador CSI implementado en MuCPP.

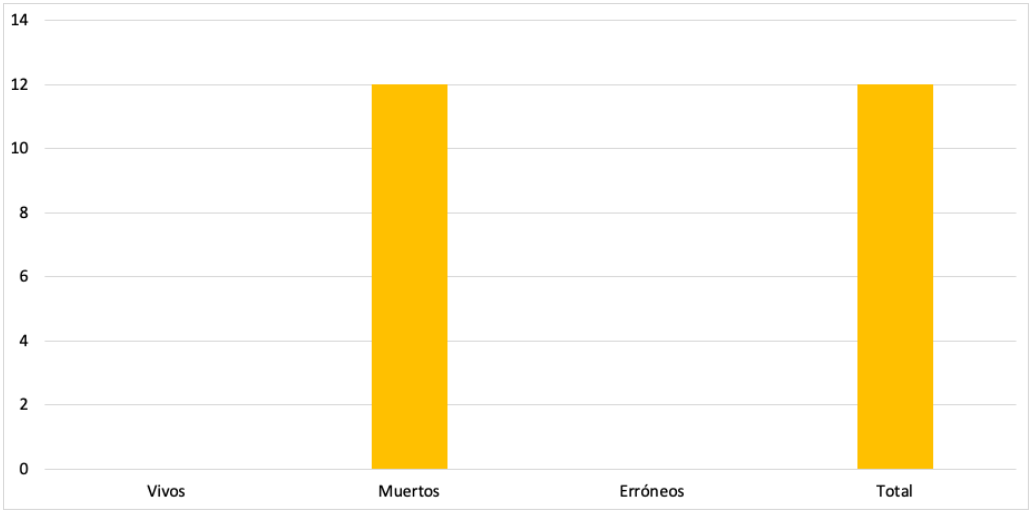

Figura 35: Gráfica salida operador CSI sobre proyecto Antonie.

Como se puede observar en la figura 35, se han obtenido un total de doce mutantes en el proyecto Antonie, de los cuales todos resultaron muertos.

Cabe destacar que este proyecto contempla una cantidad de pruebas suficientes, con una calidad buena, como para poder matar a todos los mutantes propuestos por el operador.

Discusión

Proyecto	Total	Invalidos	Vivos	Muertos	Puntuación de mutación
Corrade	0	0	0	0	N/A
Antonie	12	0	0	12	100 %

Tabla 75: Tabla operador SOR en MuCPP.

Para la realización de este análisis se ha hecho uso de los resultados obtenidos, determinando si el operador ha sido bien implementado y su utilidad.

Como se ha podido venir observando en la redacción de la presente sección, el operador implementado en MuCPP, crea un número correcto de mutantes, no contemplando ningún mutante erróneo en comparación con el resto de tipos.

Finalmente podemos destacar que el operador implementado cumple con las expectativas previstas y realiza las acciones de una manera correcta y dentro de la normalidad de un operador de estas características.

Capítulo 9

Inclusión de complementos en MuCPP

Dada la continua actualización del lenguaje de programación C++, en este Trabajo Fin de Máster también se ha implementado una funcionalidad que permite poner a disposición del usuario final la posibilidad de crear sus propios operadores de mutación e integrarlos en la herramienta MuCPP. Para la realización de esta funcionalidad se colabora con una empresa del entorno industrial que desarrolla software crítico y que tiene intención de utilizarla para crear sus propios operadores de mutación, consiguiendo así, mejorar sus pruebas de forma automática.

Por ello se busca información sobre complementos en C++. Un complemento o plug-in es una aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de una interfaz de programación de aplicaciones, también denominada *API*. Para realizar este cometido se ha realizado un análisis de la literatura disponible e investigado las diferentes opciones que permitieran hacer esto de forma dinámica y sin afectar de manera abrupta al código fuente de la propia herramienta. Entre ellas se encuentran:

- Creación de complementos mediante ficheros fuente que serían cargados por MuCPP.
- Creación de un gestor de complementos.
- Inclusión de un gestor externo de complementos.

A continuación se incluye una descripción de cada una de estas opciones.

Creación de complementos mediante ficheros fuente que serían cargados por MuCPP.

Para la implementación de esta opción se debería establecer una forma específica en la que se deberían realizar los complementos. En este caso, dado que MuCPP obtendría directamente código de los ficheros fuente, dichos operadores no se compilarían en ningún momento, por los que las mutaciones podrían no realizarse de forma correcta. Por ello, esta opción, se descarta de forma inmediata tras contemplar sus inconvenientes.

Creación de un gestor de complementos.

Para la realización de esta opción se debería crear un gestor independiente a la herramienta de gran magnitud.

Además de la creación del propio gestor, se debería realizar manuales de instalación de este gestor, dado que los complementos deberán ser compilados de forma correcta para que para obtener un adecuado funcionamiento, pudiendo dar multitud de errores con el paso de un compilador a otro.

Una vez analizado, se decide que no es la mejor opción dado que existen herramientas ya disponibles que realizan estas acciones de forma correcta, sencilla y optimizada.

Inclusión de un gestor externo de complementos.

En la actualidad existe un gran número de gestores de complementos desarrollados en diferentes repositorios, es por ello que se decide realizar una investigación sobre estos gestores para el lenguaje C++:

- **DynObj**. Es un gestor de código abierto que proporciona la posibilidad de creación de una aplicación C++ con facilidades de carga de clases en tiempo de ejecución (complementos). Está escrita en C++ estándar, y puede ser usada con cualquier compilador de C++ con buen soporte de plantillas, por lo que es multiplataforma desde el principio.

- **Pluga**. Es un gestor sencillo de complementos en C++ que es muy fácil de usar en cualquier proyecto. Esta herramienta soporta la opción de cargar complementos multiplataforma y bibliotecas compartidas en proyectos propios y la creación de una API simple y elegante para definir los complementos.

Una vez encontrados los dos gestores más conocidos para la creación de complementos en C++, se decide el uso del gestor Pluga, dado que es de código abierto y puede ser modificado si fuera necesario. Además es de fácil instalación, compilación y utilización, tal y como veremos más adelante.

9.1. El gestor de complementos Pluga en MuCPP

Para la inclusión del gestor de complementos en MuCPP, se realizan varios pasos bien definidos, los cuales son expresados a continuación y serán explicados en profundidad en secciones siguientes.

1. En primer lugar, se procede a descargar e instalar el gestor de complementos en el sistema operativo utilizado.

2. En segundo lugar, se crea una API para definir los operadores mediante complementos.

3. En tercer lugar, se modifica la herramienta en MuCPP para que acepte dichos complementos.

4. En cuarto lugar, se crea una biblioteca con código interno de MuCPP para ofrecer el uso de funcionalidades internas de MuCPP para la creación de operadores.

5. En quinto lugar, se crea un Cmake para la compilación de complementos con la extensión so.

6. En sexto lugar, se realizan múltiples pruebas, comprobando que los operadores implementados mediante complementos se ejecutan sin problema alguno.

9.2. Descarga e instalación de Pluga

Tras el estudio realizado anteriormente, se inicia la inclusión del gestor de complementos mediante la descarga e instalación del propio gestor en el sistema operativo de desarrollo.

Para realizar dicha tarea se comienza con el acceso al GitHub de pluga para acceder a información sobre el gestor y su código fuente.

Una vez realizado un estudio sobre la forma de proceder a la instalación, se realizan todos y cada uno de los siguientes pasos.

1. **Se instalan dependencias externas necesarias.**
   Se realiza la instalación de las dependencias build-essential, pkg-config, git, cmake, openssl y libss1-dev, que son necesarias para la correcta instalación y uso del gestor de complementos.

   ```
   sudo apt update
   sudo apt install build-essential pkg-config git cmake openssl libssl-dev
   ```

2. **Se clona el gestor de complementos en un directorio temporal.**
   En segundo lugar, se clona el gestor de complementos en un directorio temporal directamente desde su repositorio git, realizando el proceso mediante línea de órdenes para una descarga sencilla.

   ```
   mkdir tmp
   cd tmp
   git clone https://github.com/sourcey/libsourcey.git
   cd libsourcey

   cd src
   git clone https://github.com/sourcey/pluga.git
   cd ..
   ```

3. **Se generan los ficheros para la construcción de libsourcey con su módulo pluga.**
   Una vez descargado los ficheros fuente se procede a la construcción del Makefile para realizar el proceso de construcción de binarios posteriormente.

   ```
   mkdir build
   cd build
   cmake .. -DCMAKE_BUILD_TYPE=RELEASE -DBUILD_MODULES=OFF
       -DBUILD_MODULE_base=ON -DBUILD_MODULE_pluga=ON
   ```

4. **Construcción de los binarios.**
   Tras la generación del Makefile se procede a la construcción de los binarios necesarios para el uso del gestor.

```
make
```

5. **Instalación de las bibliotecas y sus ficheros de cabecera en el sistema.** Por último se procede a realizar la instalación de las bibliotecas y cabeceras en el sistema operativo con el fin de poder usar el gestor facilmente.

```
sudo make install
```

Una vez realizada la instalación del gestor de complementos, se comienza a implementar una API que servirá como base para la creación de complementos posteriormente por los desarrolladores.

Para ello se procede a observar cuales son las acciones necesarias para que MuCPP pueda ejecutar un nuevo operador de mutación, las cuales se expresan a continuación.

- Inicialización de nombres de operadores.
- Inicialización de atributos de los operadores.
- Inicialización de los tipos de los operadores.
- Creación del matcher de los operadores.
- Instanciación del aplicador de los operadores.

Una vez definidas las acciones a realizar por los complementos, se procede inicialmente a realizar un diagrama de conexión entre los complementos y MuCPP mediante la API.

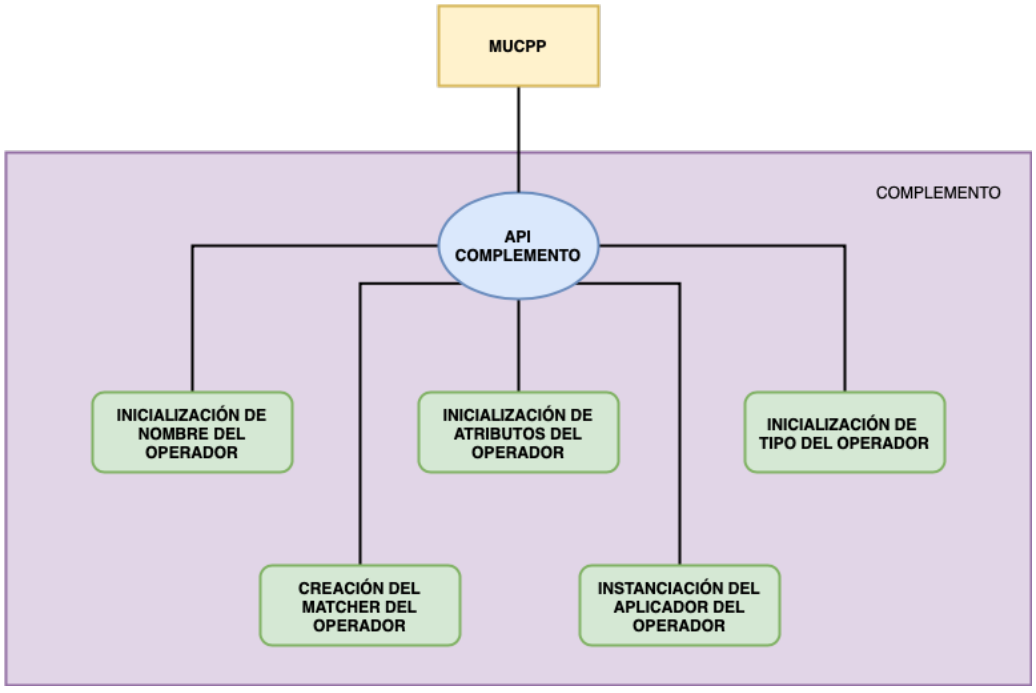

Figura 36: Conexión de MuCPP con los complementos mediante API.

Como se ha podido observar en la figura 36, se realiza una conexión directa entre MuCPP y la API del complemento, obteniendo mediante esta las diferentes funcionalidades que posee el operador. Una vez definida la estructura de los complementos, se procede a la implementación de la API.

```
// testpluginapi.h

#ifndef SCY_PluginsAPI_H
#define SCY_PluginsAPI_H


#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchersInternal.h"
#include "clang/ASTMatchers/ASTMatchersMacros.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "Mutation.h"


#include "scy/pluga/pluga.h"
#include <list>
#include <map>
#include <string>

namespace scy {
 namespace pluga {
 class IPlugin // Virtual plugin interface.
 {
 public:
     IPlugin() {};

     virtual ~IPlugin() {};

     virtual bool initializeOperatorsNames(std::list<std::string>
         &operators_names) = 0;

     virtual bool initializeAttributes(std::map<std::string,
         unsigned int> &attributes) = 0;

     virtual bool initializeTypes(std::map<std::string,
         bool> &types) = 0;

     virtual bool createMapMatchers(std::map<std::string,
         clang::ast_matchers::DeclarationMatcher> &matchs,
         std::string classPattern, std::string functionPattern) = 0;

     virtual bool apply(const clang::ast_matchers::
         MatchFinder::MatchResult &Result, ...) = 0;
  };
 }
```

```
    }

#endif
```

Tal y como se puede observar en el código anterior, se ha creado una API que contemplan las siguientes funciones:

- Constructor del complemento.
- Destructor del complemento.
- Función de inicialización del nombre del operador.
- Función de inicialización de los atributos del operador.
- Función de inicialización de tipos.
- Función de creación del Matcher del operador.
- Función de aplicación del operador.

9.3. Integración de Pluga en MuCPP

Tras la creación de la API para complementos de MuCPP mediante Pluga, se comienza con la integración del gestor de complementos en MuCPP. Para ello se procede a observar, de nuevo, cuales son las acciones necesarias para que MuCPP pueda ejecutar un nuevo operador de mutación mediante plugins, las cuales son expresadas a continuación.

- Carga de plugins.
- Inicialización de nombres de operadores.
- Inicialización de atributos de los operadores.
- Inicialización de los tipos de los operadores.
- Instanciación del matcher de los operadores.
- Instanciación del aplicador de los operadores.

Una vez reconocidas las acciones a realizar, se procede a la implementación de dichas acciones una a una.

9.3.1. Implementación de la carga de complementos en MuCPP

```
#include "PluginsOperators.h"

vector<pluga::IPlugin*> pluginsvector;


void LoadPlugins(string directorio){
    if(directorio.compare("")!=0){
     if( DIR* pDIR = opendir(directorio.c_str()) ){
            while(dirent* entry = readdir(pDIR)){
```

```
                    std::string fileName = entry->d_name;
                    if ((fileName.find(".so") != std::string::npos) && fileName!="."
                        && fileName!="..") {
                        std::string path(directorio+fileName);
                        try {
                                // Carga de operador
                                //std::cout << "Loading: " << path << std::endl;
                                SharedLibrary lib;
                                lib.open(path);
                                pluga::PluginDetails* info;
                                lib.sym("exports", reinterpret_cast<void**>(&info));

                                cout<<"Incluido plugin "<<info->pluginName<<endl;

                                if (info->apiVersion != SCY_PLUGIN_API_VERSION)
                                    throw std::runtime_error("Plugin ABI version
                                    mismatch. Expected %s, got %s.");

                                // Instanciación del plugin
                                pluginsvector.push_back(reinterpret_cast
                                    <pluga::IPlugin*>(info->initializeFunc()));

                        }
                        catch (std::exception& exc) {
                            std::cerr << "Error: " << exc.what() << std::endl;
                            assert(0);
                        }
                    }
                }
                closedir(pDIR);
            }
        }
    }
```

Como se puede observar en el código anterior, se implementa una función para la carga de complementos. Para ello se obtienen los complementos que se encuentren en la carpeta que se le ha pasado como parámetro de la función. Si existieran subcarpetas se accederían a estas de forma recursiva para localizar operadores.

En dicha función se comprueba inicialmente que existe algún complemento en la carpeta proporcionada y en caso afirmativo se carga el operador. Posteriormente, se imprime por pantalla información sobre el complemento cargado como es el nombre del operador, se inserta el complemento en un vector de complementos, del cual se obtendrán más tarde las acciones incluidas de forma sencilla.

Además de la función anterior, se crea una función para la obtención de la carpeta pasada por línea de órdenes, la cual puede ser observada a continuación.

```
void process_plugins_directory(int argc, const char **argv){
    bool found_directory = false;
    for(int narg = 0; narg<argc; narg++){
        string argumento = argv[narg];
```

```
            if (argumento.substr(0,9).compare("-plugins=")==0){
                pluginsdirector = argumento.substr(9,argumento.size()-1);
                found_directory=true;
                argv[narg]="";
            }else{
                if(found_directory==true){
                    argv[narg-1]=argv[narg];
                    argv[narg]="";
                }
            }
        }
    }
```

Como se puede observar en la función, se obtiene de los parámetros pasados por línea de órdenes el parámetro *-plugins=*, en el cual vendrá expresada la carpeta en la que se deberán buscar posteriormente los operadores implementados como complementos.

9.3.2. Inicialización de nombres de operadores

Para esta acción se ha realizado dentro del código de MuCPP un bucle de carga de la siguiente forma.

```
    //Carga de plugins

for (vector<pluga::IPlugin*>::iterator it=pluginsvector.begin();
    it != pluginsvector.end(); ++it){

        pluga::IPlugin* plugin = reinterpret_cast<pluga::IPlugin*>(*it);
        assert(plugin->initializeOperatorsNames(operators_names));
}
```

Como se ha podido observar, se recorre el vector de complementos implementado anteriormente, obteniendo de todos y cada uno de los complementos cargados la función declarada en la API para inicializar los nombres de los operadores. Con ello queda realizada la acción que se debe realizar en esta subsección.

9.3.3. Inicialización de atributos de los operadores

Para esta acción se ha realizado dentro del código de MuCPP un bucle de carga de la siguiente forma.

```
    //Carga de plugins

    for (vector<pluga::IPlugin*>::iterator it=pluginsvector.begin();
        it != pluginsvector.end(); ++it){

        pluga::IPlugin* plugin = reinterpret_cast<pluga::IPlugin*>(*it);
        assert(plugin->initializeAttributes(attributes));
    }
```

Como se ha podido observar, se recorre el vector de complementos implementado anteriormente, obteniendo de todos y cada uno de los complementos cargados la función declarada en la API para inicializar los atributos de los operadores. Con ello queda realizada la acción que se debe realizar en esta subsección.

9.3.4. Inicialización de los tipos de los operadores

Para esta acción se ha realizado dentro del código de MuCPP un bucle de carga de la siguiente forma.

```
//Carga de plugins

for (vector<pluga::IPlugin*>::iterator it=pluginsvector.begin();
    it != pluginsvector.end(); ++it){

    pluga::IPlugin* plugin = reinterpret_cast<pluga::IPlugin*>(*it);
    assert(plugin->initializeTypes(types));
}
```

Como se ha podido observar, se recorre el vector de complementos implementado anteriormente, obteniendo de todos y cada uno de los complementos cargados la función declarada en la API para inicializar los tipos de los operadores. Con ello queda realizada la acción que se debe realizar en esta subsección.

9.3.5. Instanciación del matcher de los operadores

Para esta acción se ha realizado dentro del código de MuCPP un bucle de carga de la siguiente forma.

```
//Carga de plugins

for (vector<pluga::IPlugin*>::iterator it=pluginsvector.begin();
    it != pluginsvector.end(); ++it){

    pluga::IPlugin* plugin = reinterpret_cast<pluga::IPlugin*>(*it);
    assert(plugin->createMapMatchers(matchs, classPattern, functionPattern));
}
```

Como se ha podido observar, se recorre el vector de complementos implementado anteriormente, obteniendo de todos y cada uno de los complementos cargados la función declarada en la API para la instanciación del matcher de los operadores. Con ello queda realizada la acción que se debe realizar en esta subsección.

9.3.6. Instanciación del aplicador de los operadores

Para esta acción se ha realizado dentro del código de MuCPP un bucle de carga de la siguiente forma.

```
//Carga de Plugins

for (vector<pluga::IPlugin*>::iterator it=pluginsvector.begin();
```

```
        it != pluginsvector.end(); ++it){

        pluga::IPlugin* plugin = reinterpret_cast<pluga::IPlugin*>(*it);
        assert(plugin->apply(Result, this));
    }
```

Como se ha podido observar, se recorre el vector de complementos implementado anteriormente, obteniendo de todos y cada uno de los complementos cargados la función declarada en la API para la instanciación del aplicador de los operadores. Con ello queda realizada la acción que se debe realizar en esta subsección.

9.4. Creación de biblioteca de funcionalidades internas

Una vez realizadas las acciones anteriores, se procede a la implementación de un operador mediante complementos, pero se encuentra la problemática de la falta de funcionalidades que podemos encontrar en el código interno de MuCPP, mediante funciones y clases, por lo que se decide crear una biblioteca con dichas funcionalidades para aportarla a los desarrolladores de operadores con complementos.

Para la creación de la biblioteca se crea un Makefile que compilará ficheros fuentes de MuCPP en forma de biblioteca, dicho Makefile puede ser observado a continuación.

```
CXX := clang++
RTTIFLAG := -fno-rtti
LLVMCXXFLAGS :=
    -I/usr/lib/llvm-8/include -std=c++0x -fPIC
    -fvisibility-inlines-hidden -Werror=date-time -std=c++14
    -Wall -W -Wno-unused-parameter -Wwrite-strings -Wcast-qual
    -Wno-missing-field-initializers -pedantic -Wno-long-long
    -Wno-uninitialized -Wdelete-non-virtual-dtor -Wno-comment
    -ffunction-sections -fdata-sections -O2 -fno-exceptions
    -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS
    -D__STDC_LIMIT_MACROS

CXXFLAGS := $(LLVMCXXFLAGS) $(RTTIFLAG) -fexceptions

LLVMLDFLAGS := $(shell llvm-config --ldflags --system-libs
    --libs) $(LDFLAGS)

SOURCES = mucpp.cpp auxiliary_functions.cpp git_functions.cpp
    Tests.cpp MutationMatchers.cpp OperatorConstants.cpp
    Mutation.cpp PluginsOperators.cppç

OBJECTS = $(SOURCES:.cpp=.o)

EXES = $(OBJECTS:.o=)

CLANGLIBS =
    -lclangFrontend
    -lclangSerialization
```

```
    -lclangDriver
    -lclangTooling
    -lclangParse
    -lclangSema
    -lclangAnalysis
    -lclangEdit
    -lclangAST
    -lclangASTMatchers
    -lclangLex
    -lclangBasic
    -lclangRewrite
    -lclangRewriteFrontend
    -luv

all: $(OBJECTS) $(EXES)
.PHONY: clean install
%: %.o
    $(CXX) -o $@ $< $(CLANGLIBS) $(LLVMLDFLAGS)

mucpp: auxiliary_functions.o git_functions.o Tests.o
    OperatorConstants.o Mutation.o PluginsOperators.o mucpp.o

$(CXX) -o $@ $^ $(CLANGLIBS) $(LLVMLDFLAGS)

library: auxiliary_functions.o git_functions.o OperatorConstants.o
    Mutation.o PluginsOperators.o Tests.o
    ar -rv libMutation.a auxiliary_functions.o git_functions.o
        OperatorConstants.o Mutation.o PluginsOperators.o Tests.o
    cp libMutation.a ../Creador_de_operadores/

git_functions.o Tests.o Mutation.o mucpp.o: auxiliary_functions.h
auxiliary_functions.o: git_functions.h
OperatorConstants.o: Operators.h PluginsOperators.h
PluginsOperators.o: Mutation.o
Mutation.o: OperatorConstants.h
mucpp.o: Tests.h MutationMatchers.h OperatorConstants.h
    Mutation.h PluginsOperators.h

clean:
-rm -f $(EXES) $(OBJECTS) auxiliary_functions.o git_functions.o
    Tests.o MutationMatchers.o OperatorConstants.o Mutation.o
    mucpp.o libMutation.a *~
```

Gracias al Makefile anterior, podremos obtener una biblioteca totalmente funcional con la que poder utilizar las funciones internas de MuCPP de forma transparente para el desarrollador, denominada *libMutation.a*.

9.5. Cmake de compilación de complementos

Para la creación de operadores mediante complementos se ha creado un Cmake para su compilación. A continuación se muestra dicho Cmake que ha sido creado.

```
SET( APP_NAME "Operador" )
PROJECT( ${APP_NAME} )

cmake_minimum_required(VERSION 2.6)

SET (CMAKE_C_COMPILER "/usr/bin/clang")
SET (CMAKE_CXX_COMPILER "/usr/bin/clang++")

ADD_DEFINITIONS(-I/usr/lib/llvm-8/include -std=c++0x -fPIC
    -fvisibility-inlines-hidden -Werror=date-time -std=c++14 -Wall -W
    -Wno-unused-parameter -Wwrite-strings -Wcast-qual
    -Wno-missing-field-initializers -pedantic -Wno-long-long
 -Wno-uninitialized -Wdelete-non-virtual-dtor -Wno-comment
 -ffunction-sections -fdata-sections -O2 -fno-exceptions
 -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS
 -D__STDC_LIMIT_MACROS -fno-rtti -fexceptions)

SET( LIBS
 -lclangFrontend
 -lclangSerialization
 -lclangDriver
 -lclangTooling
 -lclangParse
 -lclangSema
 -lclangAnalysis
 -lclangEdit
 -lclangAST
 -lclangASTMatchers
 -lclangLex
 -lclangBasic
 -lclangRewrite
 -lclangRewriteFrontend
 -luv
)

SET( LIBS2
 libMutation.a
)

INCLUDE_DIRECTORIES (
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${CMAKE_CURRENT_SOURCE_DIR}/libMutation
)

LINK_DIRECTORIES (
```

```
    ${CMAKE_CURRENT_SOURCE_DIR}/libMutation
)

add_library(${APP_NAME} SHARED Plugin.cpp)

target_link_libraries(${APP_NAME} libMutation.a)
TARGET_LINK_LIBRARIES(${APP_NAME} ${LIBS} )

SET(CLANG_TIDY_CHECKS "google-build-namespaces,\
google-build-using-namespace,\
google-explicit-constructor,\
google-readability-casting,\
google-runtime-int,\
modernize-avoid-bind,\
modernize-avoid-c-arrays,\
modernize-concat-nested-namespaces,\
modernize-make-shared,\
modernize-deprecated-headers,\
modernize-make-unique,\
modernize-redundant-void-arg,\
modernize-use-nullptr,\
readability-braces-around-statements,\
readability-container-size-empty,\
readability-convert-member-functions-to-static,\
readability-delete-null-pointer,\
readability-else-after-return,\
readability-implicit-bool-conversion,\
readability-isolate-declaration,\
readability-magic-numbers,\
readability-non-const-parameter,\
readability-simplify-boolean-expr,\
readability-string-compare")

set(CMAKE_CXX_CLANG_TIDY
clang-tidy;
-header-filter=.;
-checks=${CLANG_TIDY_CHECKS};
-warnings-as-errors=*;)
```

9.6. Creación de operadores mediante complementos

En la presente sección se procederá a crear paso a paso un operador de mutación mediante complementos, en este ejemplo se implementará un operador de eliminación de la sentencia de llamada a la función join de los hilos de C++. Para la realización de este ejemplo se seguirán los siguientes pasos.

1. Se recopilarán todos los ficheros necesarios para la creación.
2. Se implementará el operador de mutación mediante complementos, haciendo uso de la API.

3. Se modificará el Cmake.
4. Se compilará el complemento.

9.6.1. Recopilación de ficheros necesarios en un directorio

Para comenzar con la implementación del operador mediante complementos serán necesarios una sarie de ficheros bien definidos, y con la siguiente estructura de directorios.

```
- Directorio complemento
    CMakeLists.txt
    Plugin.cpp
    Plugin.h
    PluginsAPI.h
    - libMutation
        auxiliary_functions.h
        git_functions.h
        libMutation.a
        Mutation.h
        OperatorConstants.h
        Operators.h
        PluginsOperators.h
```

A continuación se pasa a explicar todos y cada uno de los ficheros expresados anteriormente, con el fin de un buen entendimiento por parte del lector.

- **Directorio complemento.** Directorio en el que se implementará y compilará el complemento a desarrollar.
- **CMakeLists.txt.** Fichero de compilación, mediante este fichero se podrá compilar de una forma sencilla el complemento desarrollado.
- **Plugin.cpp.** Fichero de desarrollo. En dicho fichero se encontrará el código desarrollado del complemento.
- **Plugin.h.** Fichero de cabecera del complemento. En este fichero se encontrán las cabeceras de las funciones implementadas en el fichero anterior.
- **PluginsAPI.h.** API desarrollada en secciones anteriores del presente capítulo. Mediante esta API se podrá desarrollar correctamente el complemento.
- **Directorio libMutation.** Directorio en el que se encuentran los ficheros de cabecera y la biblioteca de código interno de MuCPP para dar accesibilidad a dichas funcionalidades a los desarrolladores de complementos.
- **auxiliary_functions.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.
- **git_functions.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.
- **libMutation.a.** Biblioteca desarrollada en una sección del presente capítulo.
- **Mutation.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.

- **OperatorConstants.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.
- **Operators.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.
- **PluginsOperators.h.** Fichero de cabecera interno de MuCPP para uso de funcionalidades.

9.6.2. Implementación del operador de mutación mediante complementos

Para la realización de la presente sección se creará el complemento descrito anteriormente, del mismo modo que se venía implementando hasta el momento pero realizando todos los cambios directamente en un mismo fichero en lugar de realizarlos internamente a MuCPP.

Comenzaremos escribiendo el fichero de cabecera denominado *Plugin.h*, el cual podremos observar a continuación.

```
// testplugin.h

#ifndef SCY_Plugin_H
#define SCY_Plugin_H

#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchersInternal.h"
#include "clang/ASTMatchers/ASTMatchersMacros.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"


#include "PluginsAPI.h"
#include "Operators.h"
#include "Mutation.h"
#include "auxiliary_functions.h"

class Plugin: public scy::pluga::IPlugin{
 public:
     Plugin();
     virtual ~Plugin();

     virtual bool initializeOperatorsNames(std::list<std::string>
         &operators_names);
     virtual bool initializeAttributes(std::map<std::string, unsigned int>
         &attributes);
     virtual bool initializeTypes(std::map<std::string, bool> &types);
     virtual bool createMapMatchers(std::map<std::string,
         clang::ast_matchers::DeclarationMatcher> &matchs,
         std::string classPattern, std::string functionPattern);
     clang::ast_matchers::DeclarationMatcher CJD_Matcher(std::string
         classPattern);
     virtual bool apply(const clang::ast_matchers::MatchFinder::MatchResult
```

```
            &Result, Mutation &mutant);
        void apply_operator(const clang::ast_matchers::MatchFinder::MatchResult
            &Result, Mutation &mutant);
};


#endif
```

Como se ha podido observar, inicialmente se incluye la API de complementos creada, algunos ficheros de cabecera necesarios y posteriormente se pasa a definir las cabeceras de las funciones a implementar. Como se ha podido observar, son las mismas funciones creadas en la API, a excepción de las funciones desarrolladas específicamente para este operador, como son *CJD_Matcher* y *apply_operator*, las cuales serán utilizadas por otras funciones internas.

Por otro lado, se desarrollará el operador propiamente dicho en el fichero denominado *Plugin.cpp*, a continuación se puede observar el contenido de dicho fichero para el desarrollo de este operador. Cabe destacar que las funciones implementadas son las mostradas anteriormente en el fichero de cabecera.

```
#include "Plugin.h"
#include <iostream>


SCY_PLUGIN(Plugin, "Plugin_CJD", "0.1.1")

namespace clang {
    namespace ast_matchers {
        AST_MATCHER(clang::FunctionDecl, isImplicitFunction) {
            return Node.isImplicit();
        }
    }
}

Plugin::Plugin(){}
Plugin::~Plugin(){}

bool Plugin::initializeOperatorsNames(std::list<std::string>
    &operators_names){
    operators_names.push_back("CJD");
    return true;
}

bool Plugin::initializeAttributes(std::map<std::string,
    unsigned int> &attributes){
    attributes.insert(std::pair<std::string, unsigned int>
        ("CJD", 1));
    return true;
}

bool Plugin::initializeTypes(std::map<std::string, bool> &types){
```

```
 types.insert(std::pair<std::string, bool>("CJD", true));
 return true;
}

clang::ast_matchers::DeclarationMatcher Plugin::CJD_Matcher
    (string functionPattern){
    DeclarationMatcher matcher =
        functionDecl(matchesName(functionPattern),
            hasDescendant(
                callExpr(
                    callee(
                        cxxMethodDecl(
                            hasName("join"),
                            ofClass(
                                cxxRecordDecl().bind("CJD_Class")
                            )
                        )
                    )
                ).bind("CJD_Method")
            )
        );
    return matcher;
}

bool Plugin::createMapMatchers(std::map<std::string,
    clang::ast_matchers::DeclarationMatcher> &matchs,
    std::string classPattern, std::string functionPattern){

    matchs.insert(std::pair<std::string, clang::ast_matchers::
        DeclarationMatcher>("CJD", CJD_Matcher(classPattern)));

    return true;
}

void Plugin::apply_operator(const MatchFinder::MatchResult &Result,
    clang::Mutation &mutant){

    if (const CallExpr *FS = Result.Nodes.getNodeAs<clang::CallExpr>
        ("CJD_Method")){

        if (const Decl *FS2 = Result.Nodes.getNodeAs<clang::Decl>
            ("CJD_Class")){

            if(mutant.Rewrite.getRewrittenText(SourceRange(
                FS2->getLocation())) == "thread"){

                mutant.Operator = "CJD";

                if( (mutant.FullLocation1 = mutant.checkFullLocation
```

```
                    (FS->getBeginLoc())).isValid() ){

                    if(mutant.checkAction()){
                        mutant.replaceText("/* CJD */",
                            FS->getSourceRange());
                        mutant.createDirectory(1);
                        mutant.endOperator();
                        mutant.resetRewriter();
                    }

                    if(mutant.Action == APPLY){
                        mutant.applied = true;
                    }
                }
            }
        }
    }

    bool Plugin::apply(const clang::ast_matchers::MatchFinder::MatchResult
        &Result, Mutation &mutant){

        apply_operator(Result, mutant);
        return true;
    }
```

Como se ha podido observar en el código anterior, se ha implementado el operador de forma completa. Por un lado, se ha implementado las funciones de inicialización de nombre, de inicialización de atributos y de inicialización de tipo del operador.

Por otro lado, se ha implementado el AST Matcher de Clang, que, como se puede observar, localiza las llamadas a la función *join*. Más tarde se implementa la función de inclusión del Matcher en un vector.

Posteriormente se implementa la función de aplicación del operador, en la que, como se ha podido observar, se comprueba que el marcado realizado por el AST Matcher se corresponde con la función de la clase *Thread* de C++ y posteriormente se elimina la línea completa. Por último se inserta, mediante la función *apply* de la API la llamada a la función de aplicación del operador desarrollada.

9.6.3. Modificación de Cmake

Antes de la correspondiente compilación del operador de mutación se procede a modificar el Cmake, actualizando los datos necesarios. En este caso el Cmake quedará de la siguiente forma.

```
#SET( APP_NAME "CJD" )
PROJECT( ${APP_NAME} )

cmake_minimum_required(VERSION 2.6)

#SET (CMAKE_C_COMPILER "/usr/bin/clang")
```

```
#SET (CMAKE_CXX_COMPILER "/usr/bin/clang++")

ADD_DEFINITIONS(-I/usr/lib/llvm-8/include -std=c++0x -fPIC
    -fvisibility-inlines-hidden -Werror=date-time -std=c++14
    -Wall -W -Wno-unused-parameter -Wwrite-strings
    -Wcast-qual -Wno-missing-field-initializers -pedantic
    -Wno-long-long -Wno-uninitialized -Wdelete-non-virtual-dtor
    -Wno-comment -ffunction-sections -fdata-sections -O2
    -fno-exceptions -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS
    -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -fno-rtti
    -fexceptions)

SET( LIBS
    -lclangFrontend
    -lclangSerialization
    -lclangDriver
    -lclangTooling
    -lclangParse
    -lclangSema
    -lclangAnalysis
    -lclangEdit
    -lclangAST
    -lclangASTMatchers
    -lclangLex
    -lclangBasic
    -lclangRewrite
    -lclangRewriteFrontend
    -luv
)

SET( LIBS2
    libMutation.a
)

INCLUDE_DIRECTORIES (
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${CMAKE_CURRENT_SOURCE_DIR}/libMutation
)

LINK_DIRECTORIES (
    ${CMAKE_CURRENT_SOURCE_DIR}/libMutation
)

add_library(${APP_NAME} SHARED Plugin.cpp)

target_link_libraries(${APP_NAME} libMutation.a)
TARGET_LINK_LIBRARIES(${APP_NAME} ${LIBS} )

SET(CLANG_TIDY_CHECKS "google-build-namespaces,
```

```
        google-build-using-namespace,
        google-explicit-constructor,
        google-readability-casting,
        google-runtime-int,
        modernize-avoid-bind,
        modernize-avoid-c-arrays,
        modernize-concat-nested-namespaces,
        modernize-make-shared,
        modernize-deprecated-headers,
        modernize-make-unique,
        modernize-redundant-void-arg,
        modernize-use-nullptr,
        readability-braces-around-statements,
        readability-container-size-empty,
        readability-convert-member-functions-to-static,
        readability-delete-null-pointer,
        readability-else-after-return,
        readability-implicit-bool-conversion,
        readability-isolate-declaration,
        readability-magic-numbers,
        readability-non-const-parameter,
        readability-simplify-boolean-expr,
        readability-string-compare")

    set(CMAKE_CXX_CLANG_TIDY
        clang-tidy;
        -header-filter=.;
        -checks=${CLANG_TIDY_CHECKS};
        -warnings-as-errors=*;
    )
```

9.6.4. Compilación del complemento

Una vez realizadas todas las acciones expuestas anteriormente se procede a la compilación del complemento desarrollado. Para ello se realizan las siguientes acciones mediante línea de órdenes.

```
mkdir build
cd build
cmake ..
make
```

Una vez compilado, recuerde que el complemento puede ser utilizado mediante la opción *-plugins=DirectorioComplementos* de MuCPP.

Capítulo 10

Conclusiones y trabajo futuro

En el presente capítulo se expondrán las conclusiones a las que se llega tras la actualización de la herramienta de prueba de mutaciones MuCPP y el trabajo futuro a realizar sobre ella.

10.1. Conclusiones

Tras la realización del presente proyecto se llega a la conclusión de que MuCPP ha sido actualizado mediante la inclusión de cuatro nuevos operadores aparecidos en la literatura, gracias a los cuales, se concibe una herramienta muy completa y actualizada a los últimos estándares del lenguaje C++. Además, las conclusiones obtenidas en cada operador son buenas, generando una cantidad muy satisfactoria de mutantes y pocos mutantes erróneos.

Por otro lado, la herramienta ha sido actualizada con la inclusión de complementos para la creación de operadores de mutación, dando la posibilidad al usuario final de crear sus propios operadores de mutación de una forma sencilla e intuitiva.

Destacar que se ha aprendido mucho sobre la prueba de mutaciones, un campo en el que ya se había trabajado y en el que se pretende seguir trabajando. Se han conocido nuevos autores y se ha comparado los resultados obtenidos en los diferentes operadores con los resultados obtenidos en artículos de estos.

Añadir también que la prueba de mutaciones es un campo muy interesante dado que realiza cambios en el código de forma automática, consiguiendo así poder automatizar la creación de pruebas de software con cierto nivel de fiabilidad.

10.2. Trabajo futuro

En el trabajo futuro se encuentran diversas actuaciones sobre la herramienta de prueba de mutaciones MuCPP, los cuales son:

- Creación de una interfaz gráfica para la herramienta MuCPP, mediante la que el usuario final pueda interactuar y no deba usar la línea de comandos en ningún momento, mejorando así la usabilidad de la herramienta.

- Realización de artículos de investigación sobre la prueba de mutaciones y MuCPP, mediante los que dar a conocer las mejoras implementadas en el presente proyecto y las conclusiones obtenidas.
- Investigación sobre operadores no funcionales para la herramienta MuCPP y su inclusión en esta, haciendo la herramienta más sofisticada y con un gran número de operadores con los que mejorar el desarrollo del software.
- Conclusión de la puesta en funcionamiento de la herramienta en entornos de Industria 4.0, haciendo un análisis de su funcionamiento. Este trabajo futuro se crea con la intención de implantar generadores de pruebas automáticas en entornos industriales de la provincia o alrededores dando a conocer la herramienta.

Capítulo 11

Glosario y definiciones

11.1. Glosario

ADS: Arithmetic Operator Deletion.

AIS: Arithmetic Operator Insertion.

AIU: Arithmetic Operator Insertion.

ARB: Arithmetic Operator Replacement.

ARS: Arithmetic Operator Replacement.

ARU: Arithmetic Operator Replacement.

ASR: Short-Cut Assignment Operator Replacement.

AST: Árbol de sintaxis abstracta.

CCA: Copy constructor and assignment operator overloading deletion.

CDC: Default constructor creation.

CDD: Destructor method deletion.

CDL: Constant DeLetion.

CID: Member variable initialization deletion.

COD: Conditional Operator Deletion.

COI: Conditional Operator Insertion.

COR: Conditional Operator Replacement.

CTD: this keyword deletion.

CTI: this keyword insertion.

EAM: Acessor method change.

EHC: Exception handling change.

EHR: Exception handler removal.

EMM: Modifier method change.

EOA: Reference assignment and content assignment replacement.

EOC: Reference comparison and content comparison replacement.

IHD: Hiding variable deletion.

IHI: Hiding variable insertion.

IMR: Multiple inheritance replacement.

IOD: Overriding method deletion.

IOP: Overriding method calling position change.

IOR: Overriding method rename.

IPC: Explicit call of a parent's constructor deletion.

ISD: Base keyword deletion.

ISI: Base keyword insertion.

ISO : International Organization for Standarization .

JSD: Static modifier deletion.

JSI: Static modifier insertion.

LLVM: Low-Level Virtual Machine.

LOR: Logical Operator Replacement.

MCI: Member call from another inherited class.

MCO: Member call from another object.

OAN: Argument number change.

OAO: Argument order change.

ODL: Operator DeLetion.

OMD: Overloading method deletion.

OMR: Overloading method contents replace.

PCC: Cast type change.

PCD: Type cast operator deletion.

PCI: Type cast operator insertion.

PMD: Member variable declaration with parent class type.

PNC: New method call with child class type.

PPD: Parameter variable declaration with child class type.

PRV: Reference assignment with other comparable variable.

PVI: virtual modifier insertion.

ROR: Relational Operator Replacement.

SDL: Statement DeLetion.

SOR: Shift Operator Replacement.

TFG: Trabajo Fin de Grado.

UCA: Universidad de Cádiz.

VDL: Variable DeLetion.

11.2. Definiciones

Prueba de mutaciones. La prueba de mutaciones es una técnica de caja blanca que consiste en la introducción de pequeños fallos en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador.

Operadores de mutación. Reglas de transformación predefinidas que simulan fallos de programación habituales que son inyectados en el código fuente mediante la prueba de mutaciones.

Árbol de Sintaxis Abstracta. Estructura de datos ampliamente usada en compiladores para representar el código de un programa. Usualmente es el resultado de la fase de análisis sintáctico de un compilador.

AST Matchers Predicados para la búsqueda y acceso a los nodos del AST. Son creados mediante llamadas a funciones que permiten crear un árbol de patrones (matchers), donde los matchers internos se usan para hacer la selección más específica.

Clang: Frontend de compilador para los lenguajes de programación C, C++, Objective-C y Objective-C++.

LLVM: Infraestructura para desarrollar compiladores, escrita a su vez en el lenguaje de programación C++, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado, el tiempo de ejecución y el "tiempo ocioso" en cualquier lenguaje de programación que el usuario quiera definir.

Apéndice A

MuCPP: manual de instalación y uso

En este anexo se podrá encontrar un manuel de instalación de la herramienta de prueba de mutaciones MuCPP además de una explicación de como utilizar la herramienta.

A.1. Instalación de MuCPP

En esta sección se explica como instalar la última versión disponible de MuCPP, la cual puede ser obtenida como anexo a este documento o desde la web oficial de la herramienta en [18].

La instalación es muy sencilla, solo será necesario poseer un dispositivo con Ubuntu 18.04.3, además de esto, este sistema deberá poseer Clang-8, libclang-8-dev y git, los cuales podrán ser instalados mediante la siguiente sentencia en la terminal del sistema:

```
sudo apt install clang-8 libclang-8-dev git
```

Tras esto crearemos diferentes enlaces simbólicos para que la herramienta pueda funcionar sin ningún problema, para ello solo será necesario ejecutar las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-8 /usr/bin/clang
sudo ln -s /usr/bin/clang++-8 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-8 /usr/bin/llvm-config
```

Posteriormente se deberá configurar git, mediante las siguientes sentencias, las cuales deberán ser modificadas por el lector:

```
git config --global user.name "Nombre"
git config --global user.email "Correo"
```

Tras realizar estas acciones pasaremos a instalar la herramienta, para ello se deberá acceder a la carpeta MuCPP de la carpeta de archivos anexos:

```
cd MuCPP
```

Posteriormente se le darán permisos de ejecución a la herramienta:

```
ls -l MuCPP
chmod +x MuCPP
```

Por último se copiará la herramienta a la carpeta /usr/bin del sistema operativo:

```
sudo cp MuCPP /usr/bin
```

A.2. Funcionamiento de MuCPP

El funcionamiento de MuCPP es muy sencillo, desarrollándose en siete campos bien diferenciados, los cuales serán descritos a continuación. Toda esta información se ha obtenido de [18] y en ella se puede encontrar información complementaria sobre el uso de la herramienta o ejemplos de funcionamiento.

A.2.1. Notas previas

- Para obtener información sobre las opciones disponibles, códigos de operador, etc, debe escribir:

  ```
  MuCPP --help
  ```

- Para eliminar los mutantes creados previamente en el sistema bajo prueba y comenzar desde una copia limpia, use la opción:

  ```
  MuCPP clean
  ```

- En los ejemplos que se muestran para cada opción, se supone la existencia de una base de datos de compilación. De lo contrario, " − " debe incluirse al final de los comandos (como se explica en la sección Requisitos").

- Las opciones cuentan, analizan y aplican todas: si no se proporciona ningún archivo fuente, se procesarán los archivos en el directorio raíz (en este caso, coloque los archivos fuente de prueba en un directorio de prueba como un subdirectorio dentro del directorio raíz).

- Solo se permiten los archivos con extensión " c " , " cc ", " cxx " o " cpp ". Los archivos de encabezado serán mutados para ser incluidos en un archivo de código fuente. Si un mismo archivo de encabezado está presente en más de un archivo fuente, la herramienta evitará la creación de mutantes duplicados. Los archivos fuente proporcionados se ordenan a través de " std::sort " para que todas las ejecuciones con los mismos archivos siempre reporten los mismos mutantes.

A.2.2. Count

Mediante esta sentencia se calcula el total de mutantes en los archivos suministrados.

```
MuCPP count [file1.cpp file2.cpp...]
```

A.2.3. Analyze

Mediante esta sentencia se muestra un informe con los mutantes posibles para cada operador:

```
MuCPP analyze [file1.cpp file2.cpp ...]
```

Los informes se muestra en la pantalla, pero también se guarda en el directorio denominado " reports_analyze ", siendo estos:

- Un informe por archivo fuente analizado, con el nombre "analyze_file.txt " (donde *archivo* es el nombre del archivo de código fuente sin extensión).
- Un informe global que calcula los mutantes de todos los archivos de código fuente en su conjunto, con el nombre "analyze_MuCPP_global.txt".

La salida por pantalla es una lista con la siguiente información para cada operador de mutación:
"Operador ubicaciones atributos"
donde:

- **Operador**. Es el código del operador que corresponde.
- **Ubicaciones**. Es el número de ubicaciones donde es posible introducir una mutación de ese operador.
- **Atributos**. Número de mutaciones posibles a insertar en cada ubicación (*). Por ejemplo, si "atributos" muestra "2", eso significa que se pueden introducir dos mutantes diferentes por ubicación de mutación.

(*) Hay operadores de mutación cuyo número de atributos no es fijo, sino variable. En otras palabras, el número de mutantes para insertar en una ubicación concreta depende completamente de la ubicación dentro del código. En este caso, el atributo se marca como "1" y cada mutante producido se agrega al contador de ubicación.

A.2.4. Applyall

Mediante la siguiente sentencia se generarán todos los mutantes.

```
MuCPP applyall [archivo1.cpp archivo2.cpp ...]
```

Los mutantes se crean como ramas en el sistema de control de versiones git. Las ramas contienen una copia exacta del programa original, excepto del archivo o archivos mutados.
Formato para el nombre de mutantes:
"m + código de operador + _ + orden de ubicación + _ + atributo + _ + nombre del archivo"

Dónde:

- **Código**. Es el código del operador que corresponde.
- **Ubicación**. Es el número de ubicación del operador en el código.
- **Atributo**. Numero de la mutación a insertar en esa ubicación.

- **Nombre del archivo**. Corresponde al nombre del archivo sin el punto ni su terminación.

Notas:

- Cuando se suministra applyall, las ramas existentes se eliminan previamente.
- Los mutantes se generan a medida que las ubicaciones para mutar se encuentran en el recorrido del código.

A.2.5. Apply

Generación de un solo mutante identificado por operador, ubicación, atributo y archivo, haciéndolo mediante la siguiente sentencia:

```
MuCPP apply operador localización atributo [file1.cpp file2.cpp ...]
```

A.2.6. Run

Ejecución del conjunto de pruebas en un programa. El programa original se ejecuta cuando se suministran mutantes. En caso de que se proporcionen algunos mutantes, el conjunto de pruebas se ejecutará contra esos mutantes.

```
MuCPP run test_directory [mutant1 mutant2 ...]
```

Se genera un archivo llamado "tests_output.txt", donde cada línea es el resultado de un caso de prueba, siendo cero para una prueba satisfactoria y uno para una no satisfactoria. Si el mutante no es válido se mostrará un dos.

Si se han medido los tiempos de ejecución del conjunto de pruebas, se creará un archivo "times_output.txt", donde cada línea representa el tiempo empleado por cada caso de prueba.

Nota: Los archivos resultantes ("tests_output.txt" y "times_output.txt") se crearán en el directorio raíz y se versionarán en git.

A.2.7. Compare

Esta opción compara entre los resultados de la ejecución del conjunto de pruebas con el programa original y los resultados para los mutantes suministrados. Si no se suministran mutantes, se ejecutarán todos los mutantes existentes.

```
MuCPP compare test_directory [mutant1 mutant2 ...]
```

Este comando muestra una línea por mutante, donde para cada caso de prueba se muestra un cero para el mismo resultado, un uno para un resultado diferente y un dos en caso de un mutante no válido.

Si se midieron los tiempos de ejecución del conjunto de pruebas, se mostrarán junto a los resultados (se utiliza una " T " para separar los resultados y los tiempos).

Los resultados de la ejecución del conjunto de pruebas para los mutantes seleccionados también se recopilarán en un archivo llamado "comparsion_results.txt" en el directorio raíz cuando finalice la ejecución.

Apéndice B

Operadores individuales: instalación y uso

En este anexo encontrará un manual de instalación de los operadores individuales incluidos en las carpetas adjuntas y un manual de uso de estos operadores.

B.1. Instalación de los operadores individuales

En esta sección se explica como instalar la última versión disponible de MuCPP, la cual puede ser obtenida como anexo a este documento.

La instalación es muy sencilla, solo será necesario poseer un dispositivo con Ubuntu 18.04.3, además de esto, este sistema deberá poseer Clang-8, libclang-8-dev y git, los cuales podrán ser instalados mediante la siguiente sentencia en la terminal del sistema:

```
sudo apt install clang-8 libclang-8-dev git
```

Tras esto crearemos diferentes enlaces simbólicos para que la herramienta pueda funcionar sin ningún problema, para ello solo será necesario ejecutar las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-8 /usr/bin/clang
sudo ln -s /usr/bin/clang++-8 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-8 /usr/bin/llvm-config
```

Posteriormente se deberá configurar git, mediante las siguientes sentencias, las cuales deberán ser modificadas por el lector:

```
git config --global user.name "Nombre"
git config --global user.email "Correo"
```

Tras realizar estas acciones pasaremos a instalar el operador, para ello se deberá acceder a la carpeta con el nombre del operador deseado de la carpeta de operadores de los archivos anexos:

```
cd operadores\ individuales
cd nombre_operador_deseado
```

Tras esto, se deberá compilar el operador mediante la sentencia

```
make
```

Posteriormente se le darán permisos de ejecución al operador:

```
ls -l nombre_operador_deseado
chmod +x nombre_operador_deseado
```

Por último se copiará la herramienta a la carpeta /usr/bin del sistema operativo:

```
sudo cp nombre_operador_deseado /usr/bin
```

B.2. Uso del operador

Para realizar el uso del operador deseado solo se ejecutará la siguiente sentencia la terminal en la carpeta del archivo con el código fuente que deseamos mutar, aportando posteriormente todos los mutantes generador por el operador uno tras otro, informando de la linea y columna en la que se encuentra cada uno de ellos antes del código mutado.

```
nombre_operador_deseado nombre_archivo_codigo_fuente
```

Si lo desea, para una sencilla compilación y prueba con las pruebas realizadas en el capítulo 8, solo debe acceder a la carpeta del operador deseado y ejecutar la siguiente sentencia:

```
sh make.sh
```

Recordar que las pruebas también han sido aportadas para cada operador en la carpeta Pruebas.

Bibliografía

[1] Grupo UCASE de Ingeniería del Software - TIC025. `https://tic025.uca.es`. Último acceso: 31-08-2020.

[2] C. Sandler G. J. Myers and T. Badgett. *The Art of Software Testing, 3rd Edition. (3rd ed.).* 2011.

[3] Mike Papadakis et al. *Mutation Testing Advances: An Analysis and Survey.* Elsevier, 2017.

[4] P. Delgado-Pérez et al. *Assessment of class mutation operators for C++ with the MuCPP mutation system,.* Information and Software Technology, vol. 81, pp. 169-184, 2017.

[5] J. Lajoie S. B. Lippman and B. E. Moo. *C++ Primer, Fifth Edition. (5th ed.).* Addison-Wesley Professional, 2012.

[6] F. Palomo Lozano et al. *Fundamentos De C++. (2^a corr. y aum. ed.).* Universidad de Cádiz. Servicio de Publicaciones, 2016.

[7] Tiobe index. `https://www.tiobe.com/tiobe-index/`. Último acceso: 31-08-2020.

[8] Paul J Deitel, Abbey Deitel, and Harvey M Deitel. *C++ 11 for programmers.* Prentice Hall, 2013.

[9] A. Prieto Espinosa and B. Prieto Campos. *Conceptos De Informática.* 2005.

[10] R. S. Pressman. *Ingeniería Del Software: Un Enfoque Práctico. (7^a ed.) Mexico D.F: McGraw-Hill.* 2010.

[11] G. J. Myers et al. *The Art of Software Testing. (2nd ed.) Hoboken: John Wiley & Sons.* 2004.

[12] R. Lipton. *Fault diagnosis of computer programs, Student Report, Carnegie Mellon University.* 1971.

[13] R. G. Hamlet. *Testing Programs with the Aid of a Compiler, IEEE Transactions on Software Engineering, vol. SE-3, (4), pp. 279-290.* 1977.

[14] R. J. Lipton R. A. DeMillo and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer, Computer, vol. 11, (4), pp. 34-41.* 1978.

[15] Ali Parsai, Serge Demeyer, and Seph De Busser. C++ 11/14 mutation operators based on common fault patterns. In *IFIP International Conference on Testing Software and Systems*, pages 102–118. Springer, 2018.

[16] The LLVM Compiler Infrastructure. `http://llvm.org`. Último acceso: 31-08-2020.

[17] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Juan José Domínguez-Jiménez. Generación de mutantes válidos en el lenguaje de programación c++. *XIX Jornadas de Ingeniería del Software y Base de Datos, JISBD*, 2014.

[18] MuCPP Mutation Tool. `https://ucase.uca.es/mucpp/`. Último acceso: 31-08-2020.

[19] M. Hampton and S. Petithomme. *Leveraging a commercial mutation analysis tool for research.* 2007, DOI: 10.1109/TAICPART.2007.4344125.

[20] Parasoft insure++. `http://www.parasoft.com/products/insure`. Último acceso: 31-08-2020.

[21] A. Derezinska and K. Kowalski. *Object-oriented mutation applied in common intermediate language programs originated from C#.* IEEE, 2011, doi:10.1109/ICSTW.2011.54.

[22] Plextest, itregister. `http://www.itregister.com.au/products/plextest`. Último acceso: 31-08-2020.

[23] M. Kusano and C. Wang. *CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications.* 2013, DOI: 10.1109/ASE.2013.6693142.

[24] Testooj. `https://alarcos.esi.uclm.es/testooj3/`. Último acceso: 31-08-2020.

[25] Jumble. `http://jumble.sourceforge.net`. Último acceso: 31-08-2020.

[26] Jeff Offutt Lin Deng and Nan Li. *Empirical evaluation of the statement deletion mutation operator.* In 6th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2013), pages 80–93, Luxembourg, March 2013.

[27] C++ data types. `https://www.geeksforgeeks.org/c-data-types/`. Último acceso: 31-08-2020.

[28] Miguel Ángel Álvarez García. *PROYECTO FIN DE GRADO - Automatización de técnicas de prueba de software: Implementación de operadores de mutación y nueva funcionalidad para MuCPP.* Septiembre de 2019.